Circles and Ellipses using Polar co-ordinates
Posted on Thursday, April 18 @ 08:05:46 CEST by God

hussainweb writes: Hi, there is no need for introduction as you all must be knowing what a circle and an ellipse is. So, I will start straight with the article on how to draw them...

Circles and Ellipses using Polar co-ordinates

This article was contributed by Hussain.

Welcome to my article on drawing circles and ellipses. It is extremely simple, as you shall see.

Hi, there is no need for introduction as you all must be knowing what a circle and an ellipse is. So, I will start straight with the article on how to draw them.

Circles

So, let's start with the easy ones first. Circles are perfectly round figures. In lines you need the cartesian co-ordinate system to draw pixels easily. In the same way, you need Polar co-ordinates to draw circles.

Polar co-ordinates

All of you must be familar with the cartesian system of co-ordinates, i.e. the one in which pixel is represented in the form of (x,y). This system is used on computer screens to draw all the graphics. But for drawing circles, this system is not useful. So, we have another system in which drawing circles is

extremely easy. This system is called **Polar co-ordinate system** and each pixel is represented by (radius,angle) instead of (x,y).

By now, you must have realised how to draw a circle using polar co-ordinates. As it is represented in the form of (radius,angle), you can write code in which it increments the angle in loops and completes a circle. Isn't it simple? You know the radius of the circle that is to be drawn(you give it the radius) and angle is incremented from 0 to 360 degrees in a loop. In this way, you can draw a complete circle.

But wait, you say. Something's missing. How do you draw with Polar co-ordinates on computer screen. Don't computers use cartesian system? Yes, so you have to convert the polar co-ordinates' angle and radius to the cartesian co-ordinates' x and y co-ordinates. How? There is a formula to do that. The formula for conversion to cartesian system is given below.

```
x = Cosine(angle) * radius
y = Sine(angle) * radius
```

Doesn't it sound simple? With this formula you can complete the project of drawing circles. Here's the code in Basic but can be adapted to almost all other languages.

```
radius = 200

For angle = 0 To 360
x = Cos(angle) * radius
y = Sin(angle) * radius
PSet (x, y), Color
Next
```

Simple, isn't it? This draws a circle of radius 200 pixels. One more issue. Generally, in most languages `Cos` and `Sin` functions accept angles in radians and not our conventional degrees. So, you have to convert the angle from degrees to radians. This is simple too. You just multiply the angle with pi and divide it by 180. That's it, this will be our modified code.

```
pi = 3.14159265358979323846264338327 95

radius = 200

For angle = 0 To 360
x = Cos(angle * (pi / 180)) * radius
y = Sin(angle * (pi / 180)) * radius
PSet (x, y), Color
Next
```

OK, we fixed this issue which is prominent in most languages. But there is one more thing. The circle appears grainy. This is

because 360 degrees are too less for circles in high-resolutions. You can fix this problem in two ways. One method is to draw lines between two pixels in the circle and the other is to change the system of angle.

Let's see the first one. Drawing lines may reduce the quality of circle in high resolutions. It will not appear smooth. This way is faster than the second method, though. Let's see it.

```
pi = 3.14159265358979323846264338327950

radius = 200

PSet (radius, 0), Color

For angle = 0 To 360
x = Cos(angle * (pi / 180)) * radius
y = Sin(angle * (pi / 180)) * radius
Line -(x, y), Color
Next
```

This feature is available in few languages. One such language is Basic.

The second method produces smooth circles, but is slower than first one except in cases where circle is very small. This method involves changing the angular system from degrees to any other unit. We will define the unit and for the purposes of universally acceptable units, we will put it in such a way that the number of that units in the circle will be equal to it's circumference.

Didn't understand? I will show it by the means of code.

```
pi = 3.14159265358979323846264338327950

radius = 200
circumference = 2 * pi * radius

For angle = 0 To circumference
x = Cos(2 * angle * (pi / circumference)) * radius
y = Sin(2 * angle * (pi / circumference)) * radius
PSet (x, y), Color
Next
```

This code produces smooth circles always. We'll see the code step by step. We calculate the circumference of the circle by the formula 2 * pi * radius. This is the formula for calculating the circumference. Let's say circumference is *x*. Then, since the circumference will be equal to the number of the pixels that have to be drawn, we can use a system of units in which there are 0 to *x* degrees available in a circle. But while drawing, we need to convert that system into radians. So, we use the

formula (2 * angle * (pi / circumference)) and this will be equal to the angle in radians. Notice that if the system was 0 to 360 degrees, even then this formula would work. In fact, it was derived from that method.

We fixed a lot of issues, didn't we? Now there is only one more issue left. The circles which will be drawn will always be at the origin, so you need to translate the circles to the co-ordinates you want to use. This is simple, you just add x and y to the center of circle co-ordinates in the code.

```
pi = 3.14159265358979323846264338327 95

cx = 300: cy = 200

radius = 200
circumference = 2 * pi * radius

For angle = 0 To circumference
x = Cos(2 * angle * (pi / circumference)) * radius
y = Sin(2 * angle * (pi / circumference)) * radius
PSet (cx + x, cy + y), Color
Next
```

That's it. Now the circle drawing example is a success. You can use this code in your programs to draw high quality extra-smooth circles.

Ellipses

Now for ellipses, this is mainly based upon the circles part and only little extra introduction. Ellipses are round figures which are not perfectly round. They are oval. We can represent an ellipse in two ways, one is using eccentricities and other is using two radii which define radius along x axis and y axis. With this information, it is extremely simple to use this code.

```
pi = 3.14159265358979323846264338327 95

cx = 300: cy = 200

radius = 200
circumference = 2 * pi * radius

For angle = 0 To circumference
x = Cos(2 * angle * (pi / circumference)) * xradius
y = Sin(2 * angle * (pi / circumference)) * yradius
PSet (cx + x, cy + y), Color
Next
```

Pretty, isn't it? You can start using this code in your programs.

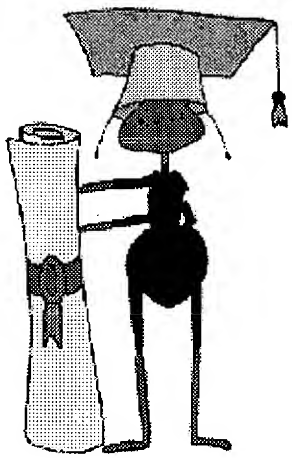
Well, that's it for now. I hope you enjoyed this. This is pretty simple, if you have any questions or doubts, please ask me. Contact info is given below.

Later, I will show you how to draw arcs and using equations to draw circles and ellipses. I will also show you how to draw ellipses using eccentricities instead of radii along x and y axis. Till then, good bye.

Oh! Yes. Please feel free to comment about this. Ask me is you have any questions. You can contact me by email at hussainweb@indiatimes.com and/or visit my website **http://hussainweb0.tripod.com**.

Bye, I hope you enjoyed my article and please rate it.

A Little Geometry The increasing capabiliti computer graphics systems has left many



A Little Geometry

The increasing capabilities of computer graphics systems has le
in the dust. It's great to be able to run dazzling application
with a "paint" program, but many of us find it difficult to des
images of our own. Becoming an artist is perhaps a bit more th
are willing to take on! It is important to remember, however,
are wonderful number-crunchers. With a little application of p
you can have the computer take on much of the work for you-- an
isn't that why we have computers in the first place?

A complete review of plane geometry is a bit beyond the scope o However, I'm going to run through some of the things I think you'll find most useful. I'd also like to suggest that you might dig out your old textbooks or rummage through your local used book store. It may have seemed like a dry subject at the time, but when you can watch the results growing on your computer screen, you will have a much better idea of how geometry can be useful to you-- and it can be surprisingly fun, too!

In geometry talk, a "point" doesn't have any actual size. In our case, we want to apply geometry to physical reality, namely the computer screen. As far as we're concerned, a "point" will be an individual graphics dot, also called a "pel" or "pixel" (for "picture element"). We can safely dispense with such formalities for our applications, for the most part.

The most important thing about a point is that it has a location! Ok, that may not seem staggering, but it happens that there are a number of ways of specifying that location. The most common method is called the Cartesian coordinate system. It is based on a pair of numbers: X, which represents the distance along a horizontal line, and Y, which represents the distance along a vertical line. Consider the CGA in SCREEN 2, for instance. It has a coordinate system where X can be 0 - 639 and Y can be 0 - 199. The points are mapped on kind of an invisible grid.

The Cartesian coordinate system makes it easy to visualize how a given point relates to other points on the same plane (or screen). It is particularly useful for drawing lines. Horizontal and vertical lines become a cinch: just change the X value to draw horizontally, or the Y value to draw vertically. Squares and rectangles (or boxes) can be formed by a combination of such lines. You can define an area of the screen in terms of an imaginary box (as GET and PUT do) with nice, clean boundaries. When we get to diagonal lines, it's a bit more of a nuisance, but still easy enough with the proper formula. That means we can do triangles too. Curves are worse... when it comes to even a simple circle or ellipse, the calculations start to get on

the messy side. For things like that, though, there is an alternative.

Another way of describing the location of a point is by Polar coordinates. In Cartesian coordinates, the location is specified by its horizontal and vertical distances from the "origin" or reference point, (0,0). In Polar coordinates, the location is specified by its distance and angle from the origin. Think of it as following a map: Cartesian coordinates tell you how many blocks down and how many blocks over the point is, whereas Polar coordinates tell you in which direction the point is and how far away it is "as the crow flies".

The Polar coordinate system is great for describing many kinds of curves, much better than Cartesian. For example, a circle is defined as all of the points at a given (fixed) distance from a center point. Polar coordinates include both a distance and an angle, and we've already got the distance, so all we need to do is plot points at all of the angles on a circle. Technically, there is an infinite number of angles, but since our points don't follow the mathematical definition (they have a size), we don't have to worry about that.

Let me digress for a moment to talk about angles. In BASIC, angles are specified in "radians". People more often use "degrees". Fortunately, it isn't hard to convert from one to the other. Both may be visualized on a circle. In radians, the sum of the angles in a circle is twice pi. In degrees, the sum of the angles is 360. That's something like this:

```
                     90 deg, 1/2 * pi rad
                           /---|---\
                          /    |    \
                         /     |     \
           180 degrees  |___   .   ___|    0 deg, 0 rad; or...
           pi radians   |             |  360 deg, 2 * pi rad
                         \     |     /
                          \    |    /
                           \---|---/
                     270 deg, 3/2 * pi rad
```

Ok, so that's a grotesquely ugly circle! Hopefully it shows the important thing, though. Angles start at zero on the extreme right and get larger as they work around counter-clockwise. The places marked on the "circle" are places where lines drawn horizontally and vertically through the center intersect the outside of the circle. These serve as a useful reference point, especially in that they help show how the angles can be construed from a Cartesian viewpoint.

So much for angles. I'll go into conversion formulae, the value of pi, and other good junk a bit later on. Right now, let's get back to our discussion of Polar coordinates.

I've explained how the Polar system makes it easy to draw a circle. Since you can vary the range of angles, it's equally simple to draw an arc. If you wanted to make a pie chart, you might want to join the ends of the arcs to the center of the circle, in which case you'd keep the angle constant (at the ends of the arc) and plot by changing the distance from zero to the radius. Circles are also handy for drawing equilateral polygons... you know, shapes with sides of equal length: triangle, square, pentagon, hexagon, etc. In

this case, the best features of the Cartesian and Polar systems can be joined to accomplish something that would be difficult in either alone.

The starting point for these polygons is the circle. Imagine that the polygon is inside a circle, with the vertices (pointy ends, that is, wherever the sides meet) touching the edge of the circle. These are equilateral polygons, so all of the sides and angles are the same size. Each of the vertices touches the circle, and each does it at exactly the same distance from each other along the arc of the circle. All of this detail isn't precisely necessary, but I hope it makes the reasoning a bit more clear!

The circle can be considered as being divided by the polygon into a number of arcs that corresponds to the number of vertices (and sides) the polygon has. Think of a triangle inside a circle, with the tips all touching the circle. If you ignore the area inside the triangle, you will see that the circle is divided into three equal arcs. The same property is true of any equilateral polygon. As a matter of fact, as the number of vertices goes up, the circle is partitioned into more, but smaller, arcs... so that a polygon with a large enough number of vertices is effectively a circle itself!

Anyway, the important thing is the equal partitioning. We know how many angles, be they degrees or radians, are in a circle. To get the points of a polygon, then... well, we already know the "distance" part, that's the same as the radius. The angles can be calculated by dividing the angles in the whole circle by the number of vertices in the desired polygon. Trying that case with the triangle, assuming a radius of 20 (why not), and measuring in degrees, that would give us the Polar points (20, 0), (20, 120), (20, 240). To make this a triangle, we need to connect the points using lines, which is easy in Cartesian coordinates. Since the computer likes Cartesian anyway, we just convert the Polar coordinates to Cartesian, draw the lines, and viola!

That's essentially the method used by the G#Polygon routines. It's very simple in practice, but I haven't seen it elsewhere... probably because people forget about the Polar coordinate system, which is what makes it all come together. Polar coordinates also have simple equations for figures that look like daisies, hearts, and other unusual things. See "Equations, Etc" and ROSES.BAS for more information.

On a side note, the Cartesian system isn't used by all computers, although it's the most common. Cartesian coordinates are the standard for what is called "raster" displays. The Polar coordinate system is used on "vector" displays. One example of a vector display that you may have seen is the old Asteroids video arcade game. They tend to be used for drawing monochrome "framework" pictures where the image must be very sharp (unlike in raster images, the diagonal lines aren't jagged, since there's no raster "grid").

In this section, I'm going to list a number of equations and so forth. Some of them will be useful to you in experimenting with Polar coordinates. Some of them provide formulae for things that are already in GRAFWIZ, but which you might like to understand better. Some of them are just for the heck of it... note that not all of this information may be complete enough for you to just use without understanding it.

One problem is... if you try to draw a circle, for instance, it will come out looking squashed in most SCREEN modes. Remember we said our points, unlike mathematical points, have a size? In most graphics modes, the points are effectively wider than they are high, so a real circle looks like an ellipse.

Another problem is that these equations are based on an origin of (0,0) which is assumed to be at the center of the plane. In our case, (0,0) is at the upper right edge, which also makes the Y axis (vertical values) effectively upside-down. This isn't necessarily a problem, but sometimes it is! Adding appropriate offsets to the plotted X and Y coordinates often fixes it. In the case of Y, you may need to subtract the value from the maximum Y value to make it appear rightside-up.

The displayed form of these equations may contain "holes", usually again because the points have a size, and/or since we try to use integer math to speed things up. If the screen had infinite resolution, this would not be a problem... meanwhile (!), getting around such problems takes fiddlin'.

There are other problems, mostly due to forcing these simplified-universe theoretical equations into practical use. It's a lot easier to shoehorn in these simple equations than to use more accurate mathematical descriptions, though... a -lot- easier. So a few minor quirks can be ignored!

With those disclaimers, here's the scoop on some handy equations.

Polar coordinates may be expressed as (R, A), where R is radius or distance from the origin, and A is the angle.

Cartesian coordinates may be expressed as (X, Y), where X is the distance along the horizontal axis and Y is the distance along the vertical axis.

Polar coordinates can be converted to Cartesian coordinates like so:

```
X = R * COS(A)
Y = R * SIN(A)
```

Angles may be expressed in radians or degrees. BASIC prefers radians. Radians are based on PI, with 2 * PI radians in a circle. There are 360 degrees in a circle. Angles increase counter-clockwise from a 3:00 clock position, which is the starting (zero) angle. Angles can wrap around: 720 degrees is the same as 360 degrees or 0 degrees, just as 3:00 am is at the same clock position as 3:00 pm.

Angles may be converted between degrees and radians as follows:

```
radians = degrees * PI / 180
degrees = radians * 180 / PI
```

The value PI is approximately 3.14159265358979. For most graphics purposes, a simple 3.141593 should do quite nicely. The true value of PI is an irrational number (the decimal part repeats forever, as near as anyone can tell). It has been calculated out to millions of decimal points by people with a scientific bent (and/or nothing better to do)!

Line Drawing:

One of the convenient ways of expressing the formula of a line (Cartesian coordinates) is:

```
Y = M * X + B
```

Given the starting and ending points for the line, M (the slope, essentially meaning the angle of the line) can be determined by:

```
M = (Y2 - Y1) / (X2 - X1)
```

The B value is called the Y-intercept, and indicates where the line

intersects with the Y-axis. Given the ingredients above, you can calculate that as:

```
B = Y1 - M * X1
```

With this much figured out, you can use the original formula to calculate the appropriate Y values, given a FOR X = X1 TO X2 sort of arrangement. If the slope is steep, however, this will result in holes in the line. In that case, it will be smoother to recalculate the formula in terms of the X value and run along FOR Y = Y1 TO Y2... in that case, restate it as:

```
X = (Y - B) / M
```

Keep an eye on conditions where X1 = X2 or Y1 = Y2! In those cases, you've got a vertical or horizontal line. Implement those cases by simple loops to improve speed and to avoid dividing by zero.

Circle Drawing:

The Cartesian formula gets messy, especially due to certain aspects of the display that are not accounted for (mainly that pixels, unlike theoretical points, have a size and shape which is usually rectangular). The Polar formula is trivial, though. The radius should be specified to the circle routine, along with the center point. Do a FOR ANGLE! = 0 TO 2 * PI! STEP 0.5, converting the resulting (Radius, Angle) coordinates to Cartesian, then adding the center (X,Y) as an offset to the result. The appropriate STEP value for the loop may be determined by trial and error. Smaller values make better circles but take more time. Larger values may leave "holes" in the circle.

Polygon Drawing:

I've already discussed that, so I'll leave it as an exercise... or of course you can examine my source code if you register GRAFWIZ! The polygon routines are in BASIC, except for the line-drawing parts.

Flower Drawing:

This sort of thing would be rather difficult to do using strictly Cartesian methods, but with Polar coordinates, no problem. Here we calculate the radius based on the angle, using something like:

```
FOR Angle! = 0 TO PI! * 2 STEP .01
```

(a low STEP value is a good idea). The radius is calculated like so:

```
Radius! = TotalRadius! * COS(Petals! * Angle!)
```

The Petals! value specifies how many petals the flower should have. If it is odd, the exact number of petals will be generated; if even, twice that number will be generated.

These figures are technically called "roses", although they more resemble daisies. Try the ROSES.BAS program to see how they look.

Other Drawing:

Experiment! There are all sorts of interesting things you can do with the Polar coordinate system in particular. Dig up those old Geometry texts or see if your Calculus books review it. If you've kept well away from math, try your local library or used book store.

E-Mail Fredric L. Rice / The Skeptic Tank

Associated Topics || Dr. Math Home || Search Dr. Math

Drawing Spirals on Your Computer

```
Date: 9/5/95 at 4:17:24
From: Anonymous
Subject: Spiral formula(s)

Hello !

I want to draw spirals with the computer. I don't know how.
Can you send me which formula(s) I can use to do it.
If possible it will be easier for me to use Cartesian formulas.
Thank you in advance.

Fabrice Bailly
```

```
Date: 9/6/95 at 12:18:1
From: Doctor Ken
Subject: Re: Spiral formula(s)

Hello!

I think the easiest way to think about spirals is in polar coordinates, and
then you can convert to cartesian coordinates in your program.  In polar
coordinates, you can have the angle just go from 0 to as big as you want
depending on how many times around you want to go, and then you can have
the radius be some function of x that gets bigger as x gets bigger.  One of the
simplest functions you could use is F(x) = x.  You could also use x^2, or
Sqrt{x}, or whatever.  If you want something funky, try F(x) = x * Sin(x),
and make sure that your Sine is in radians.

To convert from polar coordinates to Cartesian, use the conversion
x = r Cos[t]
y = r Sin[t].

-Doctor Ken,  The Geometry Forum
```

Associated Topics:
High School Equations, Graphs, Translations

Search the Dr. Math Library:

Dialog DataStar

options logoff feedback help

databases search page titles

Document

Select the documents you wish to save or order by clicking the box next to the document, or click the link above the document to order directly.

save locally as: PDF document ☐ include search strategy

order

☐ **document 1 of 1** Order Document

INSPEC - 1969 to date (INZZ)

Accession number & update
3685363, A90097970, B90051132, C90051749; 900000.

Title
Reduction of artifacts by digital filtering in Fourier image reconstruction.

Author(s)
Cheung-J-Y; Ahluwalia-B; Baik-S; Ed. by Kim-Y; Spelman-F-A.
Author affiliation
Sch of Electr Eng & Comput Sci, Oklahoma Univ, Norman, OK, USA.

Source
Images of the Twenty-First Century. Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society (Cat. No.89CH2770-6), Seattle, WA, USA, 9-12 Nov. 1989, p.356-7 vol.2.
Sponsors: IEEE.
Published: IEEE, New York, NY, USA, 1989, 6 vol. (xxiii+xviii+xiii+xv +xv+xv+2087) pp.
ISSN
CCCC: CH2770-6/89/0000-0356 ($01.00).
Publication year
1989.
Language
EN.
Publication type
CPP Conference Paper.
Treatment codes
P Practical.

Abstract
The use of digital filtering is explored as a means of minimizing artifacts in a general class of image reconstruction approaches commonly referred to as direct Fourier reconstruction methods. In the **polar-to-Cartesian conversion, applications** of windowing operations for the two-dimensional interpolation-type methods are seen to reduce artifacts according to a number of error measures. For the one-dimensional case, the use of a high-resolution spline interpolation gives the lowest error measures. The use of the chip-Z transform is proposed as a way to generate transformed projection data in a concentric square grid to eliminate part of the one-dimensional interpolation currently needed for the processing. (4 refs).

Descriptors
computerised-picture-processing; Fourier-transforms.
Keywords

image artifacts reduction; 2D interpolation methods; 1D interpolation; digital filtering; Fourier image reconstruction; **polar** to **Cartesian conversion;** windowing operations; error measures; high resolution spline interpolation; chip Z transform; concentric square grid.

Classification codes

```
A0650M    (Computing devices and techniques).
B6140     (Signal processing and detection).
C5260B    (Computer vision and picture processing).
```

Path: Home / General Engineering References/**Manufacture & Processing**/ Manufacturing Engineer's Reference Book

Never before have the wide range of disciplines comprising manufacturing engineering bee detail in one volume.

Table of Contents **Data**

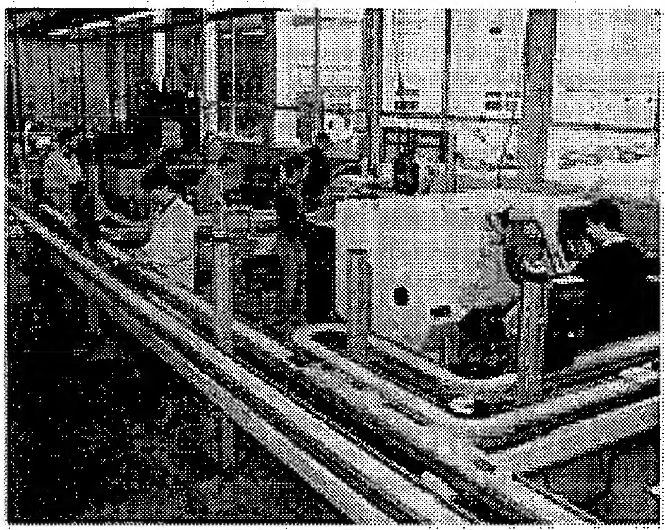


Figure 11.113 Flexible manufacturing system (Dundee Institute of Technology)

manufacturing system (FMS). The flexible manufacturing system is an integrated conglomeration of CNC machines and robots with the capability of producing a range of products with a minimum amount of human intervention. Materials handling and transportation is actioned through automated guided vehicles (AGVs), or motorised conveyors and the process starts from raw materials input through to quality control and finished product at output. Overall control of the system is governed by a central computer in a hierarchical type structure. Inventory control, procurement of materials, orders and computer aided design may also form an integral part of the overall process to be controlled.

Figure 11.113 illustrates a flexible manufacturing system, developed at the Dundee Institute of Technology. The system receives raw material direct from a continuous casting machine. In a typical application, the manufacturing route produces a range of pipe fittings which are assembled by a robot at the end of the production line.

The flexibility of the system makes it adaptable for a wide range of products. Flexibility would also normally incorporate the ability to re-route the process in the event of a failure at one of the machining workstations. *Figure 11.113* is a fairly representative example of typical robot applications in a flexible manufacturing system.

11.8.1 Robot geometry

In choosing a robot system for a range of specific tasks, some consideration should be given to the geometrical configuration of the robot system. The geometrical coordinate system relates to the relative positioning of the hand and has ramifications pertaining to the computational effort required in the position and velocity control functions.

The Cartesian geometry is typified in the gantry-type robot. These robots allow movement in three rectangular orthogonal directions and the hand motion corresponds in direct proportion to the motion along the three independent axes. The Cartesian system is by far the simplest in terms of computational effort. The gantry-type robot has much similarity with the typical overhead travelling crane.

Cylindrical polar geometries involve one rotation in combination with two linear-motion axes. The linear-motion axes are normally height h and reach r. This results in a cylindrical working envelope as shown in *Figure 11.114*.

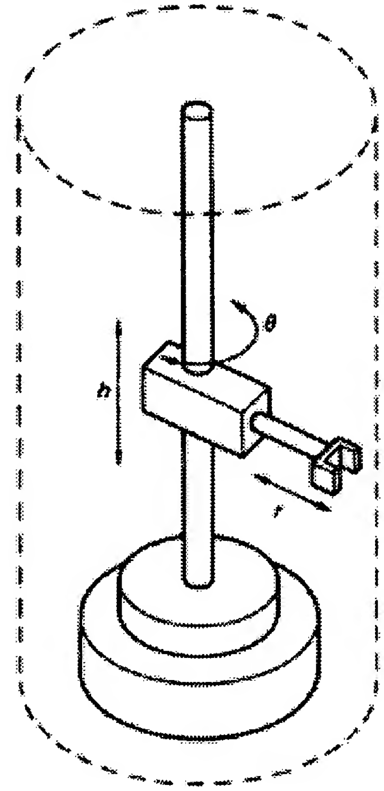


Figure 11.114 Cylindrical polar geometry

Linear hand motion across the r, θ plane involves simultaneous rotation with variable reach r. The relationship between the cylindrical polar coordinates and the Cartesian reference frame are computed from the expressions:

$$z = h \tag{11.66}$$

$$x = r\cos(\theta) \tag{11.67}$$

$$y = r\sin(\theta) \tag{11.68}$$

Spherical polar geometries involve two mutually perpendicular rotations in combination with one linear axis (*Figure 11.115*).

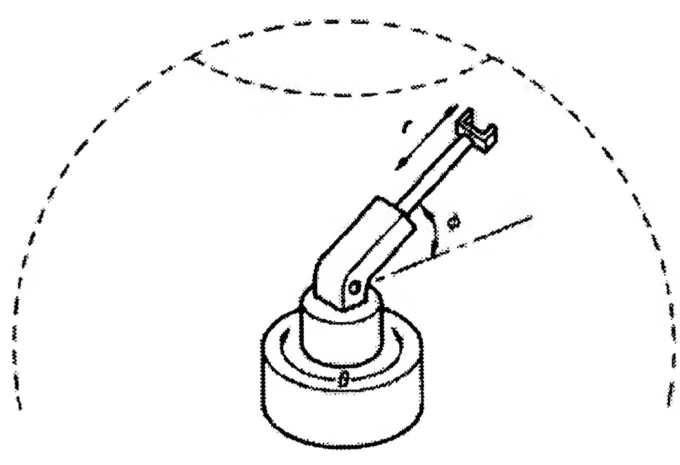


Figure 11.115 Spherical polar geometry

ESD: The Electronic System Design Magazine

October, 1988

SECTION: Vol. 18 ; No. 10 ; Pg. 95; ISSN: 0893-2565

LENGTH: 2401 words

HEADLINE: How to digitize color; Imaging & Graphics Special.; technical

BYLINE: Siegel, Shep

BODY:
How to Digitize Color

Color image processing has lagged significantly behind its monochrome counterpart, despite the rapid growth and use of digital image processing in the last decade. This disparity is due to the added price and complexity of color imaging systems.

Until now, color imaging systems were configured by "ganging" three monochrome channels. Not only do such systems cost three times as much, but any errors present in the individual monochrome channels produce group errors when the channels are combined. Recent advances in both digital and analog signal processing have helped overcome cost and complexity issues, allowing for the development of multistandard color digitizers and display controllers.

Multiformat Sensors

Color imaging sensors are available in a variety of formats beyond monochrome Vidicon tubes and charge-coupled device (CCD) sensors. Even if only TV-type devices are considered, many formats still exist, and, because the output format from the camera and the sensor need not be the same, it is important to understand the translation (transcoding) that may take place between the sensor and the camera's output to the digitizer. Available sensor formats include: prism-split GBR, striped YPrPb, and striped GBR. The analog signals from these sensor topologies may be output directly, transcoded to a different format, or encoded to produce an NTSC or PAL composite video output. Common analog television formats include: monochrome, component, and composite video. Monochrome video, either RS-170 (60 Hz) or CCIR (50 Hz), has been the primary standard of the image processing community. For color imaging, a monochrome camera combined with a color wheel can be used to sequentially digitize green, blue, and red (GBR) components. Those signals can then be transcoded into other formats, if desired. This format does not contain any direct bandwidth limitations, and some sensors may produce signals with fidelity beyond 5 MHz.

Component video is a parallelization of three monochrome channels such that co-sited green, blue, and red samples are output. No bandwidth penalty exists with three identical monochrome sensors. However, some striped CCD cameras have internal analog transcoders to provide GBR output. In those cases, the output resolution will be no greater than that which the sensor may provide. Regardless of the sensor, the component GBR format provides a way to interconnect and process color signals by a threefold replication of a single monochrome channel. Although this method does not compromise any signal bandwidth, it may not be the most efficient domain for later digital signal processing and storage.

Component video refers to component analog video (CAV) in several flavors. Here, a monochrome luminance signal, Y, is isolated from the two color difference signals, Pr and Pb. This lets the system process the Y information at full bandwidth, while using less bandwidth for the color information. The concept of augmenting a high-resolution luminance signal with a lower-resolution chrominance signal is common in television. It is based on the psychovisual phenomenon that intensity detail is more easily detected than spatial chromatic detail. In component video, Y, Pr, and Pb refer to the analog components where the luminance (Y) signal is accompanied by two orthogonal color vectors (Pr, Pb). This is similar to the NTSC (Y,I,Q) and PAL (Y,U,V) formats, in that Pr/Pb, I/Q, and U/V are all orthogonal color vectors.

Composite video is the most common video formats used in television broadcast systems. NTSC is made up of a luminance signal and a chrominance signal modulated on a color subcarrier. In the NTSC system, the two rectangular color modulation components are termed I and Q; in the PAL system, they are termed U and V. Composite video is a bandwidth compromise, as a segment of the luminance spectrum is used for the color subchannel. This subcarrier frequency (3.58 MHz in NTSC, 4.43 MHz in PAL) is quadrature-modulated by the rectangular color components and then added to the luminance signal, by an encoder, to form a composite signal. Composite video can be decoded back into its component forms by a decoder; however, the bandwidth stripped in the encoding process is difficult, if not impossible, to recover.

S-VHS video is a mix of composite and component technologies. An S-VHS signal (as sent down the standard 4-pin miniature DIN Y/C connector) contains two video signals: a component luminance signal (Y), and a modulated chrominance signal (C). This represents a good compromise between the component and composite formats: the Y signal has no bandwidth compromise, and monochrome information may pass through the system at full bandwidth.

When recorded or reproduced from tape, these signals remain on their own channels. This allows the full-luminance bandwidth to be preserved from image processor to tape and back again without encoder or decoder degradation. As higher-resolution television develops, versions of these common formats will emerge with additional bandwidth.

Prior to performing any color digital image processing, the input analog signal must be processed in several domains. Time-domain stability is important to reduce group errors, and amplitude transcoding and bandwidth limiting may be required. Specifically, the required processing depends on the analog input format and the desired digital format. In all cases an appropriate antialiasing filter should be selected to prevent out-of-band components from being digitized.

Amplitude Domains

Three common amplitude domains exist. GBR is a "tri-mono" representation where luminance and color information are encoded among the green, blue, and red components. YPrPb is a "rectangular" color representation where color is represented in Cartesian coordinates and augmented with an independent luminance (Y) signal. Finally, hue-saturation-value (HSV) is a "polar" color representation where color is represented in polar coordinates: Saturation represents the distance, r, from the origin, and hue, the angle. This color information is augmented with an independent luminance value (V).

In general, a method is needed to transcode between any of these formats. The equations for these standard color transformations are given below. First, the relationship between GBR space and a common color difference space are given as:

The simple geometric **conversions** between **Cartesian** (rectangular) and **polar** coordinates, and back, can be shown to be: Saturation v P.sub.r.sup.2 + P.sub.b.sup.2 Hue = tan.sup.-1 P.sub.b / P.sub.r P.sub.r = Saturation x cos(Hue) P.sub.b = Saturation x sin (Hue)

By combining amplitude transcoding with selective bandwidth limiting, it is possible to digitally sample the color signal in many ways. Common digital sampling formats include GBR, YCrCb, and HSV.

GBR 4:4:4 expresses the full-bandwidth sampling of each G, B, and R component. The nomenclature "4:4:4" comes from the broadcast industry, where a common sampling rate is "4 times color subcarrier." While other sampling rates have become common, the practice of assigning the number 4 to represent a reference sample rate has remained. Thus, GBR 4:2:2 describes a sampling format, where the blue and red signals are sampled at half the rate of the green signal.

YCrCb 4:2:2 is a full-bandwidth luminance system and a half-sample rate color system. This format has significant features that make it desirable. Although it has less overall bandwidth than GBR 4:4:4, it has the same luminance bandwidth. The lower bandwidth chroma channels seldom prove a limitation, except with studio-quality GBR input. Because of the lower bandwidth, less memory is used. Only two "frames" are required instead of three, as the picture is now made up of one frame of luminance and two half-horizontal resolution (hence half-size) frames of color components.

HSV 4:2:2 is sampled the same way as YCrCb 4:2:2, except that the color components are placed in polar space. This may make them easier to process for some machine-vision algorithms.

YCrCb 2:1:1 is a low-bandwidth color channel where the luminance is reduced to half bandwidth, the chroma to quarter. While signals in this format may appear "soft," they have the advantage of packing an entire color image into the memory or processing channel occupied by a single "Monochrome 4" coded channel. This attribute makes the format desirable where memory and image storage are significant factors.

In all the aforementioned formats, the signal is assumed to be sampled to full-converter precision, nominally 8 bits. As with monochrome digitizers, linearity in the converter is an important consideration. The differential gain and linearity specifications become more important in a color system, as each channel represents one part of a group signals.

Time-domain distortions can result from a non-ideally sampled analog signal. The two principal sources of time-domain distortion in color imaging systems are time-base errors and group-delay errors. While time-base errors include sample clock jitter and skew, group-delay errors are primarily a function of the input prefilter. If compensation is not made for both types of errors, measurable distortion of the signal will result.

It is desirable in many imaging systems to have a separate clock for the A/D converter, other than the system clock. This is because many devices have an A/D clock rate that differs widely from a multiple of the system clock. However, when the two signals are similar, or even seemingly identical, it has been common practice to use a single phase-locked loop (PLL) for both the system and A/D clocks.

Scanline Variations

When digitizing a signal from a mechanical device, such as a video tape recorder (VTR), there will likely be scanline-to-scanline variations in the horizontal period. Some lines will be

longer, some shorter, but by the end of the frame, the errors will cancel out. These time-base errors are due to mechanical rotation of the VTR head, which may speed up or slow down, briefly altering the horizontal period.

Left uncorrected, these variations can cause errors of as much as plus-or-minus 25%. The instantaneous variations in horizontal period produced may, at best, cause an error in the digitization of the signal, and at worst, cause the system's main PLL to unlock. By designing a specialized PLL to track the input signal, the A/D may be fed a clock that accurately corresponds to equal-length time segments in the input. The converter output, then fed to a FIFO, will absorb the line-to-line time differences between the A/D clock and system clock. This circuit, referred to as a time-base corrector (TBC), allows an accurate, solid lock to mechanical recording devices.

Group-Delay Error

Group delay refers to the delay/frequency distortions that occur when phase shift is not linear with frequency. The main causes of group-delay error, as associated with image processing systems, are the analog filters used to prefilter and reconstruct the analog waveform before and after digitization. Specifically, the prefilter (or antialiasing filter) should be carefully specified, as any error introduced prior to the converter will be difficult to remove and will adversely affect the image quality.

Aside from the obvious effects of poor image quality, group-delay errors may be particularly noticeable in systems designed to inspect images. It is common for an unequalized filter to have a delay envelope of over 100 nsec within its passband. This is enough to move a "slow," low-frequency edge an entire pixel position with respect to a "fast," high-frequency edge. Clearly, this would detrimentally affect any application trying to resolve to, or beyond, pixel position.

Group delay can be measured by sending a frequency sweep through the filter and measuring the delay time as a function of frequency. Ideally, the delay time will remain constant. However, this is not the case for designs such as elliptic filters. But by adding delay equalizer stages to the filter design, these errors can effectively be reduced.

A standard television test signal, termed Multipulse, was designed for the testing of group-delay error in television systems. Multipulse consists of five sine-squared pulses modulated with five discrete frequencies. By measuring the system output to the Multipulse stimulus, frequency response and group delay can be observed.

Color Image Processing

To address the needs of system integrators wishing to add color capability to machine-vision and pattern-recognition systems, Datacube has developed its Digicolor VME-based board. As part of the MaxVideo series of image processors, the board can be used to digitize color from NTSC or PAL signals and to use the data for further processing among MaxVideo modules. Figure 1 shows an example of the color image processing that can be performed with the Digicolor and the Warper Mark II three-board set.

To overcome the group delay and synchronization problems inherent in most color frame-grabber designs, Datacube has incorporated both a composite video encoder/decoder and a timebase corrector on-board. Digitally, three sampling modes are used on the board. This allows the GBR 4:4:4, HSV 4:4:4, HSV 4:2:2, HSV 2:1:1, YPrPb 4:2:2, and YPrPb 2:1:1 formats to be digitized at full video speed to and from the Digicolor board.

In machine-vision and pattern-recognition applications, images are often captured and the

information in them is filtered to binary data where a decision can be made. In such configurations, adding a color channel, particularly as a subchannel running parallel to an existing monochrome system, can add a useful extra metric by which to inspect the scene.

Applications have also existed in pictorial databases, but rarely in the realm of image processing. Adding the computational power (which is taken for granted in most monochrome imaging systems) to a color picture database allows greater functionality. For example, image processing hardware can be used to assist in the compression and expansion of the images during storage and retrieval. Enhancement algorithms that are common to monochrome imaging may be adapted to color image processing, although not easily and directly.

GRAPHIC: Chart; Original image and three variations allowed by Digicolor VME board. (chart)

SIC: 3672 Printed circuit boards

IAC-NUMBER: IAC 07065797

IAC-CLASS: Computer

LOAD-DATE: August 11, 1995

prev **Document 7 of 7**

Mathematical Functions

QuickDraw GX provides mathematical functions for

- fixed-point operations on `Fixed`, `long`, and `fract` number formats
- fixed-point operations on a `wide` number format
- vector operations
- Cartesian and polar coordinate point conversions
- random number generation
- linear and quadratic roots
- bit analysis

A description of each QuickDraw GX mathematics function is provided in the section "Mathematical Functions" beginning on page 8-42.

Operations on Fixed, long, and fract Numbers

QuickDraw GX provides functions that perform operations on `Fixed`, `long`, and `fract` number formats. Functions are provided that

- determine the product of two numbers (a b)
- determine the quotient of two numbers (a / b)
- determine the product of two numbers and the quotient of a third number (a b) / c
- determine both the sine and cosine of an angle measured in degrees [sine(angle) and cosine(angle)]
- determine the square root of a number (a)*1/2*
- determine the cube root of a number (a)*1/3*
- determine the magnitude of a two-dimensional vector

The functions that perform operations on `Fixed`, `long`, and `fract` number formats are described in the section "Fixed-Point Operations" beginning on page 8-42.

Operations on wide Numbers

QuickDraw GX provides functions for operations on `wide` numbers. Functions are provided that

- determine the sum of two `wide` numbers (a + b)
- determine the difference between two `wide` numbers (a - b)

- determine the product, as a `wide` number, of two `long` numbers (a b)
- determine the quotient, as a `long` number (without remainder), of a `wide` number divided by a `long` number (a / b)
- determine the result, as a `long` quotient and a `long` remainder, of dividing a `wide` number by a `long` number (a / b + remainder)
- determine the square root of a `wide` number (a)*1/2*
- negate a `wide` number (-a)
- shift bits in a `wide` number to the right or left
- determine the highest order bit in the absolute value of a `wide` number
- compare two `wide` numbers

The functions that perform operations on `wide` number formats are described in the section "Operations on wide Numbers" beginning on page 8-49.

Vector Operations

QuickDraw GX provides vector operation functions that

- determine the dot product of two vectors (v*1* v*2*)
- determine the dot product of two vectors and divide by a number (v*1* v*2*)/a

The use of QuickDraw GX vector operation functions is described in the section "Performing Vector Operations" beginning on page 8-29. These functions are described in the section "Vector Operations" beginning on page 8-54.

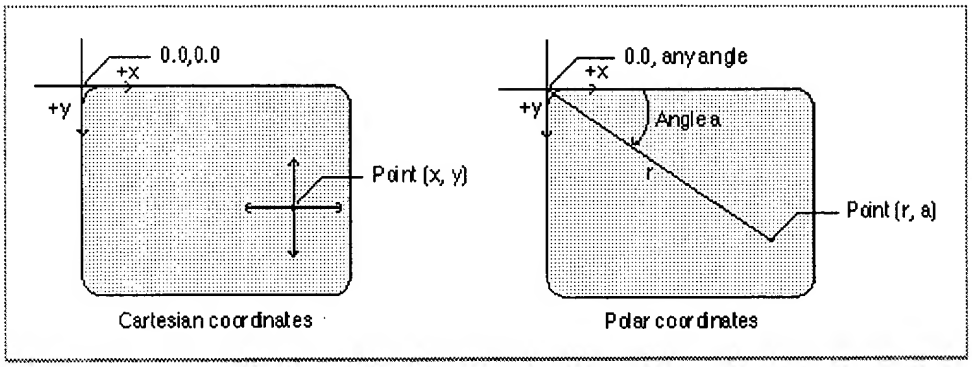
Cartesian and Polar Coordinate Conversion

You use Cartesian coordinates to specify points with QuickDraw GX. Some shapes, such as rectangles, are more easily drawn using Cartesian coordinates; however, some shapes that have symmetry about a point are more easily drawn with polar coordinates. For that reason, QuickDraw GX provides conversion routines so that you can work in either coordinate system.

For QuickDraw GX, **Cartesian coordinates** have a positive x direction to the right and a positive y direction downward (not upward, as in many other Cartesian coordinate systems). Cartesian coordinates are written in the order (x, y). The origin is at (0, 0). The `gxPoint` structure describes points using Cartesian coordinates.

Polar coordinates have the same origin point as Cartesian coordinates, but locations are specified differently. The polar coordinate of a point is specified by the length of the radius vector r from the origin to the point and the direction of the vector is specified by polar angle a. Angles in QuickDraw GX are measured clockwise in degrees from the Cartesian coordinate positive x-axis. The polar coordinate of a point specified by a vector of length r and direction a degrees from the x-axis is written as point (r, a). The polar origin point has the coordinates $(0, a)$, where a is any angle. Points having polar coordinates are defined by the `gxPolar` structure. The `gxPolar` structure is described in the section "Constants and Data Types" beginning on page 8-35. The relationship of the Cartesian and polar coordinates is shown in Figure 8-1.

Figure 8-1 Cartesian and polar coordinates



The `gxPolar` location (r, a) corresponds to the `gxPoint` location ($r \cos(a)$, $r \sin(a)$). The mathematical relationship between the two coordinate systems is given by the expressions $r2 = x2 + y2$ and $\tan(a / 2) = y / (r + x)$. The angle can also be defined by the more familiar term $\tan(a) = y / x$.

The use of the polar-to-Cartesian and Cartesian-to-polar coordinates functions are described in the section "Converting Between Cartesian and Polar Coordinates" beginning on page 8-29. These functions are described in the section "Cartesian and Polar Coordinate Point Conversions" beginning on page 8-56.

Random Number Generation

The QuickDraw GX random-number algorithm generates random integers in the range of 0 to 2*count* - 1, where *count* is the number of bits to be generated by the random number generator.

The sequence of values that the random number generator produces is dependent upon the initialization value called the **seed**. The algorithm uses the seed to calculate the next random number and a new seed. If no seed is provided, QuickDraw GX uses a default seed value of 0. To repeat a sequence of random numbers, you can use the same seed value.

QuickDraw GX provides random number generation functions that

- generate a sequence of random bits
- change the seed used by the random number algorithm
- determine the current seed for the random number algorithm

The use of the random number generation functions is described in the section "Generating Random Numbers" beginning on page 8-33. These functions are described in the section "Random Number Generation" beginning on page 8-58.

Roots of Linear and Quadratic Equations

QuickDraw GX provides mathematical functions that

- determine the root of a linear equation
- determine the roots of a quadratic equation

The linear and quadratic equation solving functions are described in the section "Linear and

Quadratic Roots" beginning on page 8-60.

Bit Analysis

QuickDraw GX provides a mathematical function that allows you to determine the highest bit number that is set in a number.

The `FirstBit` function is described in the section "Bit Analysis" beginning on page 8-62.

Previous | Book Contents | Book Index | Next

Robert Penner's Programming Macromedia Flash MX
by Robert Penner
ISBN: 0-07-222356-1
Price: **$49.99 US**
Available July 2002
Buy Now

Chapter 3

Math 1: Trigonometry and Coordinate Systems

Math is essentially relationships. We all know that relationships make the world go round. An equation may look complex, but it's simply a way of describing a relationship. The precise relationships of science have Math as their native language.

It is also the framework of beauty. As I write this, I can see a water fountain shooting 30 feet into the air, cascading down around itself and sending hundreds of thin rings on a slow journey across a still pond. The scene is gorgeous in its own right, but appears all the more beautiful because I can see the equations behind the movement. The parabola in the curve of the falling water. The sine waves in the ripples on the pond. The angles of incidence and reflection in the wavering mirror of the birch trees.

Math is power. Power to re-create the water fountain in a computer animation. Power to understand and control behavior. Power to solve a riddle, given a few clues. Power to interpret the past, create the present, and predict the future. We all use math on a regular basis to fill gaps in our knowledge—probably more than we realize. Our minds are constantly crunching numbers to get us to appointments on time, or leave a tip that's big (or small) enough. Math is an essential part of the problem-solving process. If you can convert a piece of information into a number, it is easier to work with in a programming context.

For the ActionScript animator, the subjects of trigonometry and coordinate systems are fundamental. They allow you to work with *space* with confidence and precision. This chapter will introduce these topics, laying out the concepts and providing encapsulated ActionScript code you can plug into your projects.

Trigonometry

Though it may seem intimidating, trigonometry is simply the measuring of triangles ("tri-gon" is Greek for triangle). The triangle is one of the simplest two-dimensional

shapes possible. Take any three points on a plane, connect them with lines, and you have a triangle. You may remember a few special triangles—the *equilateral* and *isosceles* triangles, for instance—but in animation one is particularly useful: the *right triangle*.

The Right Triangle

A right triangle has one 90-degree angle, or "right angle." Figure 3-1 shows a generic right triangle, with the right angle marked by a small square in the corner. The horizontal and vertical sides of the triangle are labeled *a* and *b*, respectively. The longest side, *c*, is called the *hypotenuse*.

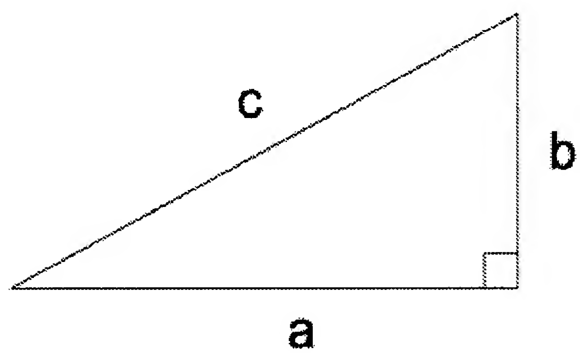


Figure 1: A right triangle with sides *a*, *b*, and *c*

Modern civilization was built on the right triangle. Our space is filled with rectangles and cubes, each composed of right angles. Our squared-off boxes, buildings and streets reveal our obsession with the perpendicular. The 90-degree angle symbolizes order and precision. The mathematical relationships within the right triangle, between its three sides and three angles, are foundational for the architect, the engineer, and the ActionScript animator.

The Pythagorean Theorem

Most students learn this famous equation in junior high school. The Greek mathematician Pythagorus discovered a relationship between the three sides of a right triangle. By taking the squares of the short sides, *a* and *b*, and adding them together, you can find the square of the hypotenuse *c*.

Pythagorus' Theorem:

$c^2 = a^2 + b^2$

If we want to solve for *c*, we take the square root of both sides of the equation:

$c = \mathrm{sqrt}(a^2 + b^2)$

Let's explore this equation with an example from ordinary life. We know intuitively that "cutting a corner" gives us shorter trips and saves time. If you wanted to walk from one corner of a football field to the other, cutting straight across would obviously be faster than walking along the sides. But how much time do you actually save?

If the field is 40 meters wide and 100 meters long (see Figure 3-2), your trip would be 140 meters if you walked along the sides *a* and *b*. On the other hand, you could cut through the middle, along line *c*.

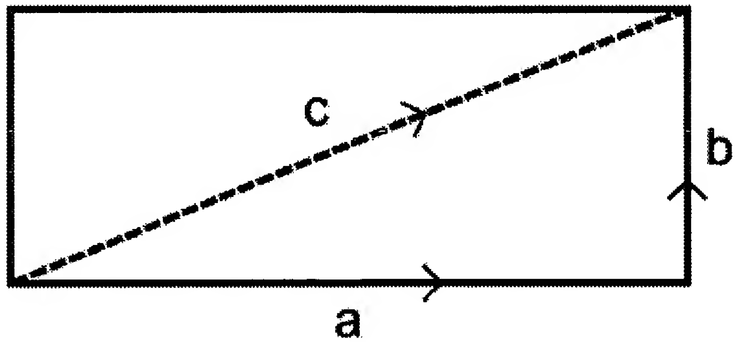


Figure 2: Cutting across a field

The lines *a*, *b*, and *c* form a right triangle. To find the value of *c*, we use the Pythagorean Theorem:

$c^2 = a^2 + b^2$
$c = \text{sqrt}(a^2 + b^2)$
$c = \text{sqrt}(100^2 + 40^2)$
$c = \text{sqrt}(10000 + 1600)$
$c = \text{sqrt}(11600)$
$c = 107.7$

If you cut across the field, along line *c*, your trip is about 108 meters. Compared to walking along sides *a* and *b*, you saved yourself 32.3 meters of walking, or 23 percent. You would see the biggest percentage difference with a square field (of any size). In this scenario, cutting across diagonally saves about 41 percent. If you're interested, you can derive this number yourself as an exercise, using the Pythagorean theorem.

Distance Between Two Points

A common task in ActionScripted animation is to find the distance between two points. Figure 3-3 shows two arbitrary points, with the coordinates (x_1, y_1) and (x_2, y_2). The Pythagorean Theorem can be used to find this distance if a right triangle is constructed. First, draw a line *d* between the two points, followed by the appropriate horizontal line *dx*, and vertical line *dy*. Side *d* is now the hypotenuse of a right triangle, and its length is the distance between the two points.

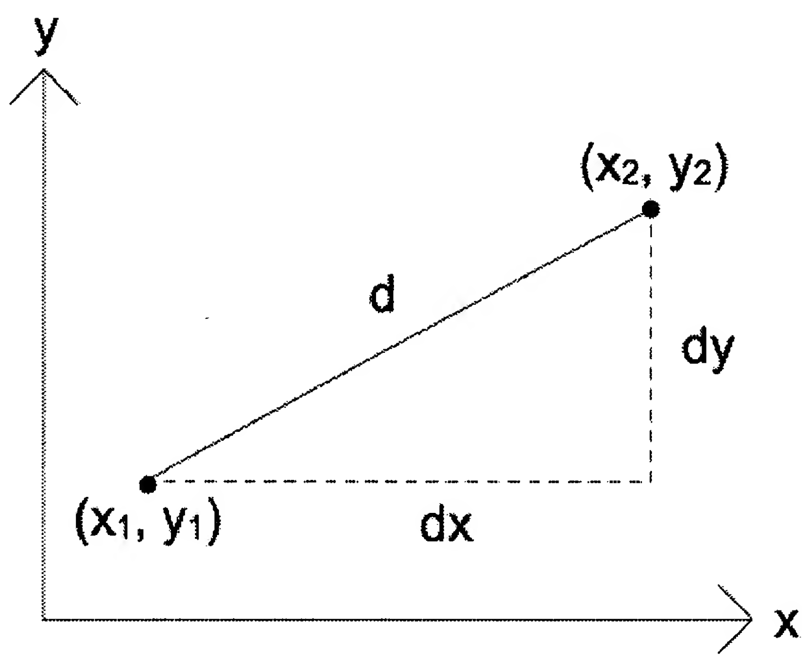


Figure 3: Finding the distance between two points

Our distance equation will have this form:

$d = \text{sqrt}(dx^2 + dy^2)$

Translated into ActionScript, the code reads like this:

```
d = Math.sqrt (dx*dx + dy*dy);
```

We can easily write a function to return the distance based on *dx* and *dy*:

```
function distance (dx, dy) {
   return Math.sqrt (dx*dx + dy*dy);
}
```

However, this setup is not as convenient as we might like. The parameters *dx* and *dy* must be calculated before calling the function. What we really want is to take the coordinates for any two points (x_1, y_1) and (x_2, y_2), pass them to a function, and receive back the distance between the points. Here is such a function, *Math.distance()*:

```
Math.distance = function (x1, y1, x2, y2) {
   var dx = x2 - x1;
   var dy = y2 - y1;
   return Math.sqrt (dx*dx + dy*dy);
};
```

The calculation of *dx* and *dy* has been moved inside the function, which saves steps. The function has also been placed in the *Math* object, where it can be easily accessed from any timeline.

Angles in Right Triangles

So far, we've explored the Pythagorean relationship between the three sides of a right triangle. We will soon look at more interesting connections between triangle sides. But

first, we must turn our attention to the *angles* of the right triangle. A triangle has three angles, which always add up to 180 degrees. In a right triangle, one of the angles is 90 degrees. Thus, the other two angles must add up to 90 degrees.

Radian Measurement of Angles

Degrees are the most common unit of measurement for angles in everyday life. There are 360 degrees in a circle, but this number is completely arbitrary. The radian unit was developed by mathematicians as a standard, metric way to measure angles. Radians are based on π(pi), a property inherent to circles. Computer programming languages typically work with angles in radians. JavaScript and ActionScript have an identical built-in *Math* object, which calculates sines and cosines in radians, not degrees.

Note: The following is a mathematical definition of a radian: When two lines extend from the center of a circle to its edge, an angle and an arc are subtended between them. The angle between the two lines is *one radian* when the length of the subtended arc exactly equals the radius.

Converting from Degrees to Radians

How does one convert from degrees to radians? The basic relationship is that πradians equals 180 degrees—a ratio of π: 180. Thus, there are 2πradians in 360 degrees, a full circle. The conversion equation is:

radians = degrees * q / 180°

In ActionScript, we can create a function to perform this conversion:

```
Math.degreesToRadians = function (angle) {
   return angle * (Math.PI / 180);
};
```

The *Math.degreesToRadians()* function accepts an angle in degrees as a parameter. It converts the angle to radians and returns the result. The following code demonstrates how to use this function:

```
// test Math.degreesToRadians()
angleDegrees = 180;
angleRadians = Math.degreesToRadians (angleDegrees);
trace (angleRadians) // output: 3.14159265358979
```

Note: In the *Math.degreesToRadians()* function, I put parentheses around (Math.PI / 180) for a reason. When Flash publishes a SWF, it compiles the ActionScript into bytecode. During that process, it evaluates whatever constant values it can to optimize the file for performance. For instance, Math.PI is always replaced with the numerical value 3.14159265358979; it is not looked up dynamically at run time. The parentheses around (Math.PI / 180) cause the compiler to evaluate the expression to 0.0174532925199433. This eliminates the need for the division to occur at run time, and thus improves performance.

Converting from Radians to Degrees

Converting angles in the opposite direction, from radians to degrees, requires a simple inversion of the ratio. The angle is multiplied by 180 / π:

```
Math.radiansToDegrees = function (angle) {
   return angle * (180 / Math.PI);
}
```

```
// test the Math.radiansToDegrees() function
trace ( Math.radiansToDegrees (Math.PI) ); // output: 180
```

Sine

At their essence, sine and cosine are simply *ratios*. A ratio is a mathematical relationship where one number is directly proportional to another. Sine and cosine describe relationships between the sides of a right triangle. In many scenarios, the sine ratio deals with the *height* of a right triangle, and cosine governs the *width*. There are more formal definitions of sine and cosine you may have heard, but here's a little thought experiment to help develop an intuitive understanding.

Imagine you have your arm flat on a table, holding a piece of string in the tip of your hand. Raising your hand changes your arm's angle with the table, and swings your hand in an arc. Meanwhile, the string hangs straight down from your hand. A triangle is formed between the table, your arm, and the string (see Figure 3-4).

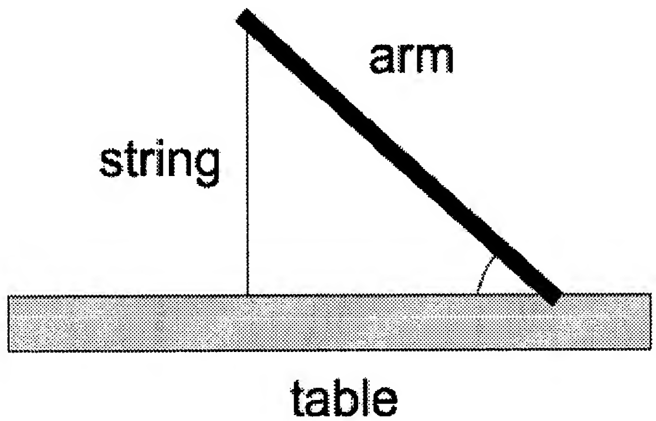


Figure 4: Exploring the sine ratio with your arm and a length of string

The length of the string changes as you change the angle of your arm relative to the table. The string gives a measurement of your hand's height above the table. The question is: how high will your hand be (that is, how long is the string) for a particular angle? Zero degrees is easy: the height is zero because your hand is flat on the table. It's also simple to find the angle at 90 degrees. The string is as long as your arm, so if your arm is three feet long, the tip of your hand is three feet above the desk.

What about 45 degrees, though? When the angle is half of 90 degrees, is the string half the length of your arm—1.5 feet long? Looking at the string, you should be able to see that this is not the case. In fact, the angle that would give you a height of half your arm is 30 degrees.

Here's where sine comes in: the sine of 30 degrees is 0.5. And for our other angles, the sine of 0 degrees is 0, and the sine of 90 degrees is 1. Are you beginning to see how sine might be related to the angle and the string? Think back to when your arm is straight up and down. Your arm is three feet long, and so is the string. The angle is 90 degrees, and the sine of the angle is 1. Remember how I said that sine is a ratio? Well, the ratio of the string to your arm length is 1, at 90 degrees.

To find the length of the string at a certain angle, you take the length of your arm and multiply by the height ratio for that angle:

string length = arm length * height ratio for angle

A sine is basically a height ratio for a given angle:

height ratio for angle = sine of angle

Thus, we can change the first equation by replacing the height ratio with the sine:

string length = arm length * sine of angle

We can test this last equation with our example. We start with an arm length of 3, and an angle of 0 degrees:

string length = 3 * sine of 0°

Since we know that the sine of 0 degrees is 0, we can see that the equation will be this:

string length = 3 * 0
string length = 0

With an angle of 90 degrees, the sine is 1, so the equation is:

string length = 3 * sin 90°
string length = 3 * 1
string length = 3

Those are fairly simple cases. Now let's try a 30-degree angle. If you have a scientific calculator (hint: there's one on your computer), you can punch in 30 and hit sin to find the sine of 30 degrees, which is 0.5. As a result, our equation is:

string length = 3 * sin 30°
= 3 * 0.5
= 1.5

Now, with this equation, we will be able to see how long the string is at 45 degrees. If you take the sine of 45 degrees on your scientific calculator, you'll find that it is approximately 0.707. Plugging that value into our equation:

string length = 3 * sin 45°
= 3 * 0.707
= 2.121

Formal Definition of Sine

So far, I have presented sine in an informal fashion. It is important to develop an intuitive understanding of the sine ratio, so you can apply it in real-world situations. We will now round out our discussion of sine with a look at its more formal definition:

sin q = opposite / hypotenuse

In our right triangle, side *b* is opposite from the angle *q*, and the hypotenuse is *c*, see Figure 3-5.

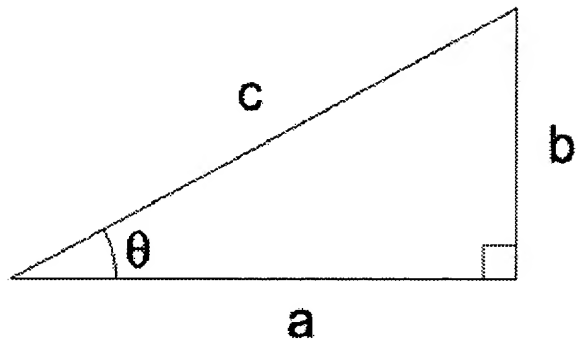


Figure 5: A right triangle with angle *q*

Thus, we can rewrite the equation as:

sin q = b / c

ActionScript – Math.sin()

The syntax to find the sine of an angle in ActionScript is simply this:

```
Math.sin (theta);
```

The variable theta is an angle in radians. Here is a simple example of how to use *Math.sin()*:

```
// test Math.sin()
trace ( Math.sin (0) ); // output: 0
trace ( Math.sin (Math.PI) ); // output: 1.22460635382238e-16 (approx. 0)
```

Note: Because of the way floating-point numbers are stored in binary format, small rounding errors appear in calculations. As a result, you will sometimes get values like 1.22460635382238e-16 when you were expecting zero. The decimal number is approximately 1.22 divided by 10^{16}, or 0.000000000000000122. Thus, it is extremely close to zero.

I prefer to work in degrees wherever possible, so I wrote a new function for the *Math* object to help me out. The following *Math.sinD()* function takes an angle in degrees and returns its sine:

```
Math.sinD = function (angle) {
   return Math.sin (angle * (Math.PI / 180));
};
```

Math.sinD() is a *wrapper function*. It wraps around *Math.sin()*, providing a different interface that better suits our purposes. The following code demonstrates how to take the sine of various angles in degrees:

```
// test Math.sinD()
trace ( Math.sinD (0) ); // output: 0
trace ( Math.sinD (90) ); // output: 1
trace ( Math.sinD (180) ); // output: 0
```

Cosine

Similar to sine, cosine is a ratio between two sides of a right triangle. However,

whereas sine is generally a height ratio, cosine is a width ratio. Returning to our example with the arm and string, imagine there is a light directly above your arm, casting a shadow onto the desk beneath it. To find the length of this shadow for different angles, we call upon cosine. Figure 3-6 shows how we can explore cosine in this scenario.

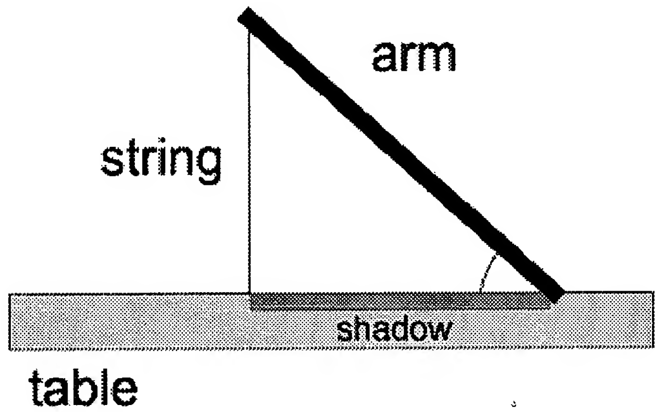


Figure 6: Exploring the cosine ratio with your arm and the width of shadow

Let's start with an easy case again. Think about how long your arm's shadow would be when your arm is straight up and down. At 90 degrees, there isn't really a shadow, so the length is zero. Without knowing anything more about cosine, could you hazard a guess as to what the cosine of 90 degrees is? If you came up with zero, give yourself a gold star—the cosine of 90 degrees is, in fact, zero. How about when your arm is flat on the desk—what would the shadow's ratio be then? Since the shadow is as long as your arm, the ratio between the two is simply 1:1. The angle of elevation is 0 degrees, and the cosine of 0 degrees is 1, as you might have guessed.

As a final exercise, let's look at 45 degrees. Remember that earlier, we found the sine of 45 degrees to be 0.707 (approximately). If you take the cosine of 45 degrees with a scientific calculator, you'll find that it is also 0.707. Plugging this value into our equation, we get:

shadow length = 3 * cos 45°
shadow length = 3 * 0.707
shadow length = 2.121

We can see that at 45 degrees the string and the shadow are the same length—2.121 feet—because the sine of 45 degrees is equal to the cosine of 45 degrees. This makes a lot of sense when you consider that the triangle between your arm, the string, and the shadow is actually half of a square. The sides of a square are equal, and the diagonal line is always at 45 degrees.

The Formal Definition of Cosine

Cosine is formally defined as "adjacent over hypotenuse," where "adjacent" refers to the side of the right triangle *closest* to the angle:

cos q = adjacent / hypotenuse

In our right triangle (see Figure 3-5), side *a* is adjacent to the angle *θ*, and the hypotenuse is *c*.

Thus, we can rewrite the equation like this:

cos q = a / c

ActionScript – Math.cos()

The syntax to find the cosine of an angle in ActionScript is similar to finding the sine:

```
Math.cos (theta);
```

The variable theta is an angle in radians, naturally. Here is a simple example of using *Math.cos()* to take the cosine of various angles in radians:

```
// test Math.cos()
trace ( Math.cos (0) ); // output: 1
trace ( Math.cos (Math.PI / 2) ); // 6.12303176911189e-17 (approx. 0)
```

Here, as before, I would prefer to specify the angle in degrees rather than radians. I wrote a *Math.cosD()* function for this purpose:

```
Math.cosD = function (angle) {
   return Math.cos (angle * (Math.PI / 180));
};
```

The following code demonstrates how to take thet cosine of various angles in degrees:

```
// test Math.cosD()
trace ( Math.cosD (0) ); // output: 1
trace ( Math.cosD (90) ); // output: 6.12303176911189e-17 (approx. 0)
```

Tangent

We have explored two trigonometric ratios: sine, which often governs height, and cosine, which often governs width. *Tangent*, the third fundamental ratio in trigonometry, governs slope.

Slope

As the angle of a line changes, so does its steepness. We can express steepness mathematically, in a ratio called *slope*. You may remember the expression "slope equals rise over run" from high-school math. *Rise* means the vertical change in position; *run* means the horizontal change. Thus, slope is calculated by dividing *vertical change* by *horizontal change*:

slope = *rise* / *run* = *vertical change* / *horizontal change*

We can transfer this concept to our familiar right triangle (see Figure 3-1). The slope of the hypotenuse *c* is the ratio of *b* to *a*:

slope of *c* = *b* / *a*

The Slope of a Line at a Given Angle

There is a particular relationship between the slope of the hypotenuse *c* and the adjacent angle *q*. This relationship is captured in the tangent ratio. The tangent of an angle is the slope of a line at that angle:

slope of *q* = *tangent* of *q*

Given the angle of a line, you can find the line's slope by using the *tan* function on a

scientific calculator. The slopes of some common angles are listed in Table 3-1:

q	tan q	Explanation
0°	0	A line at 0 degrees is horizontal, so its slope is 0.
30°	0.577...	The exact value is 1/sqrt.
45°	1	A 45-degree line balances the horizontal and vertical, so its slope is 1.
60°	1.732...	The exact value is sqrt.
90°	¥	A line at 90 degrees is straight up and down, so its slope is not a finite number.
180°	0	A 180-degree line is horizontal, so its slope is also 0.

Table 1: Values of Tangent for Common Angles

The Formal Definition of Tangent

Tangent is formally defined as "opposite over adjacent." *Adjacent* is the side of the right triangle closest to the angle, and *opposite* is the side farthest away.

tan q = opposite / adjacent

In our right triangle (Figure 3-5), side *b* is opposite to the angle *q*, and side *a* is adjacent.

Thus, we can rewrite the equation like this:

tan q = b / a

ActionScript – Math.tan()

You can calculate the tangent of an angle in Flash using the ActionScript function *Math.tan()*. The syntax is simple:

```
Math.tan (angle);
```

Again, the angle must be given in radians. I wrote my own function *Math.tanD()* that lets me specify the angle in degrees:

```
Math.tanD = function (angle) {
   return Math.tan (angle * (Math.PI / 180));
};
```

The following code demonstrates how to take the tangent of various angles in degrees:

```
// test Math.tanD()
trace ( Math.tanD (0) ); // output: 0
trace ( Math.tanD (45) ); // output: 1
trace ( Math.tanD (90) ); // output: 1.63317787283838e+16 (approx. infinity)
```

Inverse Tangent

We've seen that the tangent operator turns an angle into a slope ratio. How do we perform the reverse operation? We can take the *inverse tangent* of a slope to turn it into the corresponding angle. Inverse tangent is usually written as $tan^{-1}\,q$, but it makes occasional appearances as *arctan q* or *atan q*.

ActionScript – Math.atan()

There are two functions which calculate the inverse tangent in ActionScript. The first, *Math.atan()*, takes a single number as a parameter:

```
angle = Math.atan (slope);
```

The returned angle is in radians. Before calling *Math.atan()*, you would usually calculate *slope* by dividing *y* by *x*:

```
slope = y / x;
angle = Math.atan (slope);
```

Finding the correct angle with *Math.atan()* is a complicated process. The inverse tangent of any number can be one of two angles. For instance, tan^{-1} can be 45 degrees or 225 degrees. In general, two angles 180 degrees apart will have the same slope and tangent:

$tan\ q = tan(q + 180°)$

However, *Math.atan()* will only return one of these angles. Consequently, determining the correct angle requires a cumbersome "quadrant-checking" procedure. On top of that, the tangent of 90 degrees and 270 degrees is infinity because of a division by zero; this can be difficult to manage in code. For these reasons, I avoid *Math.atan()* altogether and employ the more user-friendly *Math.atan2()*.

ActionScript – Math.atan2()

The *Math.atan2()* function takes two parameters, *y* and *x*:

```
angle = Math.atan2 (y, x);
```

What's truly beautiful about *Math.atan2()* is that it returns the correct angle every time, no fuss, no muss. Quadrant checking is done internally, using the *x* and *y* parameters.

Since *Math.atan2()* returns an angle in radians, I wrote my own function *Math.atan2D ()* to calculate degrees instead:

```
Math.atan2D = function (y, x) {
   return Math.atan2 (y, x) * (180 / Math.PI);
};
```

The following code demonstrates how to take the inverse tangent of various angles in degrees:

```
// test Math.atan2D()
trace ( Math.atan2D (7, 7) ); // output: 45
trace ( Math.atan2D (0, -4) ); // output: 180
trace ( Math.atan2D (-6, 0) ); // output: -90
```

Note: Be careful to put *y* first, then *x*. Ordered pairs are usually written (*x*, *y*), but this is an exception.

Finding the Angle Between Two Points

The *Math.atan2()* function returns the angle of a line between a point (x, y) and the origin (0, 0). But how do we calculate the angle of a line between *any* two points (x_1, y_1) and (x_2, y_2)? We simply subtract the x and y coordinates of the two points to form a new point ($x_2 - x_1$, $y_2 - y_1$). This new point can now be passed to *Math.atan2()* to find the angle.

I wrote a function called *Math.angleOfLine()* that takes x and y coordinates for two points, and returns the angle between them:

```
Math.angleOfLine = function (x1, y1, x2, y2) {
   return Math.atan2D (y2 - y1, x2 - x1);
};
```

For convenience, the returned angle is in degrees.

In the following code example, *Math.angleOfLine()* is used to find the angle of a line between the points (3, 3) and (5, 5):

```
// test Math.angleOfLine()
theta = Math.angleOfLine (3, 3, 5, 5);
trace (theta); // output: 45
```

Note: *Math.angleOfLine()* depends on the custom function *Math.atan2D()*, so be sure to include it in your movie.

Inverse Cosine

Taking the cosine of an angle produces a ratio between -1 and 1. To work backwards, from the ratio to the angle, we take the *inverse cosine* of the ratio. In ActionScript, the process looks like this:

```
// first take the cosine
cosRatio = Math.cos (angle);
// reverse the process to find the original angle
angle = Math.acos (cosRatio);
```

The *Math.acos()* function takes a ratio between -1 and 1 and returns an angle in radians. I wrote a *Math.acosD()* function to produce degrees instead:

```
Math.acosD = function (ratio) {
   return Math.acos (ratio) * (180 / Math.PI);
};
```

The inverse cosine is commonly used to find the angle between two vectors, as we will see in Chapter 4.

Inverse Sine

Similar to inverse cosine, the inverse sine is used to find an angle. The difference is that the inverse sine operates on a sine ratio instead:

```
sinRatio = Math.sin (angle);
// in reverse
angle = Math.asin (sinRatio);
```

The *Math.asin()* function takes a ratio between -1 and 1 and returns an angle in radians. I wrote a *Math.asinD()* function to produce degrees instead:

```
Math.asinD = function (ratio) {
   return Math.asin (ratio) * (180 / Math.PI);
};
```

Compared to the trigonometric functions, inverse sine isn't used very often. I have included it for the sake of completeness.

We have now established a base knowledge of trigonometry, the study of triangles. We have learned the fundamental relationships between angles and sides in a right triangle. These include the Pythagorean Theorem, as well as the sine, cosine, and tangent ratios, and their inverses. Several practical applications of trigonometry have been explored, such as finding the distance between two points, and finding the angle of a line. Finally, we accumulated a library of functions that encapsulate different operations of trigonometry. These include wrapper functions for several built-in *Math* methods, that allow us the convenience of using degrees instead of radians—for example, *Math.sinD()* instead of *Math.sin()*. This knowledge is a good foundation for moving ahead into custom coordinate systems, and in later chapters, vectors.

Coordinate Systems

We have all encountered graphs and grids, whether in school or graphics programs. Each graph has a coordinate system, which defines how things are measured on its grid. As ActionScript animators, we need to understand different coordinate systems in order to properly render graphics. Additionally, we can define more advantageous coordinate systems and convert from one system to another.

Cartesian Coordinates

The Cartesian coordinate system is a widely used grid. Most line graphs we see, whether for the stock market or mathematical functions, use the Cartesian system.

The X and Y Axes

The grid has two axes, *x* and *y*. The *x* values increase to the right of the grid, and the *y* values increase towards the top. The location of a point on the grid is given as an ordered pair (*x*, *y*). The two axes intersect at (0, 0); this point is called the *origin* (see Figure 3-7).

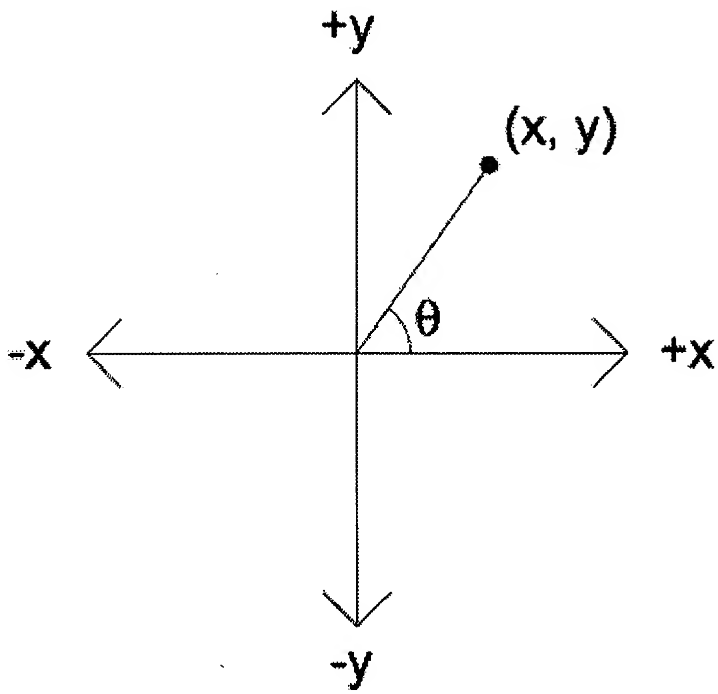


Figure 7: The Cartesian coordinate system

Grid Quadrants

The *x* and *y* axes split the Cartesian grid into four quadrants, numbered from 1 to 4, which start at the upper right and work counter-clockwise. Figure 3-8 indicates the numbering of the quadrants:

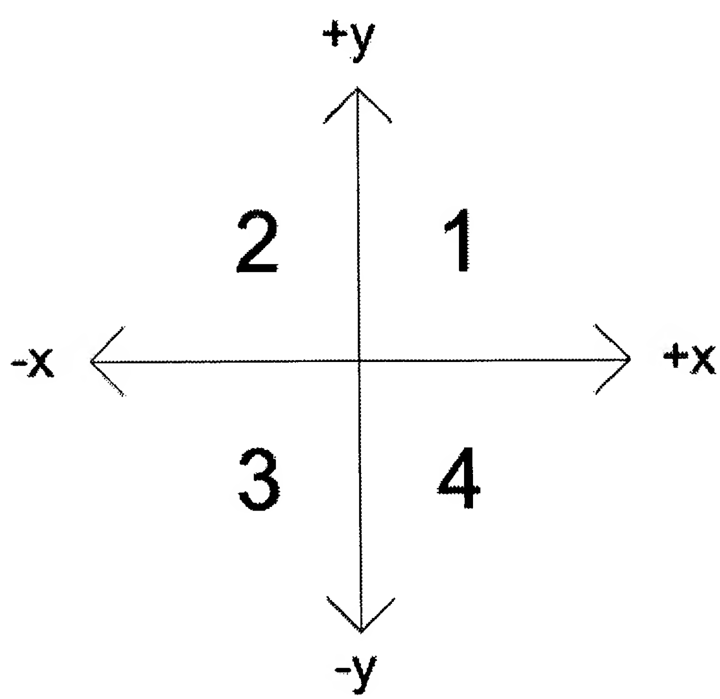


Figure 8: The four quadrants of the Cartesian grid

The Cartesian quadrants have several properties, which are summarized in Table 3-2:

Quadrant	**Corner**	***x* Values**	***y* Values**	**Angle Range**
Quadrant 1	Upper right	+	+	0° to 90°
Quadrant 2	Upper left	-	+	90° to 180°
Quadrant 3	Lower left	-	-	180° to 270°
Quadrant 4	Lower right	+	-	270° to 360°

Table 2: Properties of the Cartesian Quadrants

Angle Measurement

When angles are measured in Cartesian coordinates, the starting point is the x-axis. From there, angles are measured in a counter-clockwise direction (see Figure 3-8).

Flash Coordinates

Similar to the Cartesian system, Flash has coordinates along the *x* and *y* axis. Each movie clip has an _x property which stores its horizontal position relative to the origin (0, 0), and a _y property which stores its vertical position. The origin of a movie clip is called its *registration point*, and is denoted by a small cross. The origin of the *_root* movie clip is at the upper-left corner of the main stage (see Figure 3-9).

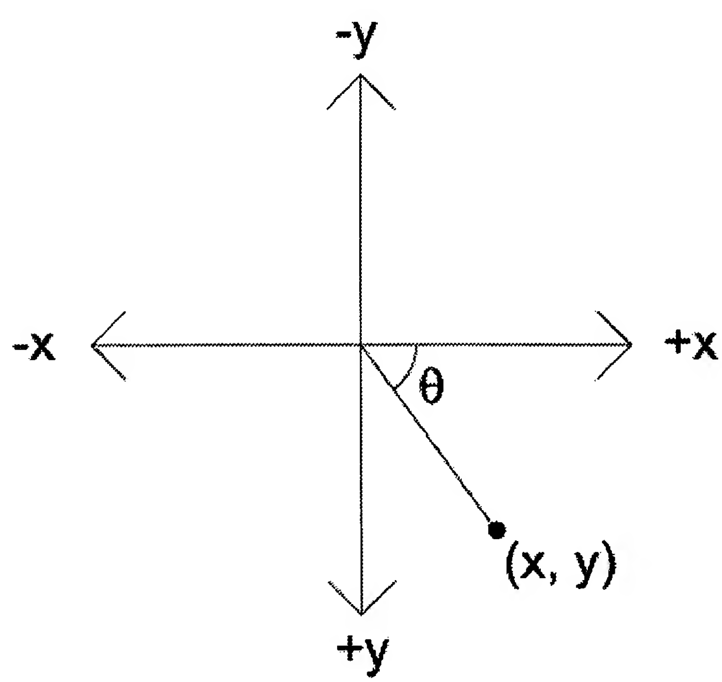


Figure 9: The Flash coordinate system

The _x property increases to the right of the screen, the same direction as the

Cartesian x. Thus, no conversion is necessary for the _x property:

Flash _x = Cartesian x

The _y property, on the other hand, increases in value towards the bottom of the screen. This is directly opposite to the Cartesian *y* coordinate, which increases towards the top (Figure 3-9). The Cartesian *y* needs to be inverted (multiplied by -1) before being assigned to _y:

Flash _y = -(Cartesian y)

Angle Measurement: the _rotation Property

All movie clips have a _rotation property which stores their angle of rotation. These angles are measured differently in Flash than on the Cartesian grid. They are measured in a clockwise direction, the opposite of the Cartesian system (see Figure 3-8 again). Thus, 10 degrees in Cartesian is -10 degrees in Flash.

Flash _rotation = -(Cartesian angle in degrees)

Also, Flash stores the _rotation property in degrees, so radian angles will have to be converted before being assigned to movie clips.

Flash _rotation = -(Cartesian angle in radians) * 180 / p

On a final note, the _rotation property always returns a number between 180 and -180. In mathematical terms:

-180 £ _rotation £ 180

However, you can set _rotation to any number you want. For example, you can give a movie clip a rotation of 370 degrees:

```
mc._rotation = 370;
```

The movie clip will be rendered at the correct angle. However, the _rotation property will internally store 10 degrees, which is equivalent to 370 degrees. If we check the _rotation value immediately after setting it to 370, we find it has been changed to 10, which lies between -180 and 180:

```
trace ( mc._rotation ); // output: 10
```

Standardizing an Angle

When doing certain calculations in ActionScript, I may need to ensure that an angle falls in the "standard" range between 0 and 360 degrees. For instance, 380, 740, and -340 degrees are all equivalent to 20 degrees (380 - 360 = 20, -340 + 360 = 20, etc.). I wrote a *Math.fixAngle()* function that will take an angle in degrees and return the equivalent standardized angle between 0 and 360 degrees:

```
Math.fixAngle = function (angle) {
   angle %= 360;
   return (angle < 0) ? angle + 360 : angle;
};
```

In the first line of code inside *Math.fixAngle()*, the *modulo* operator is used to set angle to the remainder of a division with 360. This step forces angle to fall between -360 and 360. Our goal is an angle between 0 and 360 (that is, a positive number). Thus,

we need to check if angle is negative at this stage. If it is less than zero, we add 360 to convert it to the equivalent positive angle.

Note: The internal code may be difficult to follow, simply because the function is highly optimized for speed, and uses abbreviated syntax. However, the beauty of a good function is that you don't have to understand the internal workings of the "black box." As long as you know what to put in, and what comes out, you'll be fine.

The following example demonstrates how to standardize angles using *Math.fixAngle()*:

```
// test Math.fixAngle()
trace ( Math.fixAngle (740) ); // output: 20
trace ( Math.fixAngle (-340) ); // output: 20
trace ( Math.fixAngle (20) ); // output: 20
```

The normalized angle for all three input angles (740, -340, and 20) is 20.

Converting from Cartesian to Flash Coordinates

I prefer to think in Cartesian coordinates as much as possible. When I want to code complex dynamic movement in Flash, as in my *Black Star* or *Sphere Burst* experiments, I start by developing the concept on a Cartesian grid—either mentally or on paper. Animations such as these usually require conversions into polar coordinates (discussed later in this chapter) and back again. At some point, movie clips will be moved around by changing _x, _y, or _rotation. These conversions are easiest if I stay in Cartesian coordinates as long as possible. However, I will eventually need to render the graphics in Flash coordinates.

When it comes to the ActionScript implementation, I find it best to store the positions of objects in x, y, and rotation *variables*, which are separated from the x, y, and rotation *properties*. This gives me the flexibility to perform my calculations in the Cartesian coordinate system. When it comes time to render these coordinates to the screen, I convert to Flash coordinates, with the following code:

```
// convert Cartesian coordinates to Flash coordinates
// x, y, and rotation are existing number variables
// and mc is a movie clip instance
mc._x = x;
mc._y = -y;
mc._rotation = -rotation;
```

The x variable doesn't need to be changed during the conversion. However, y and rotation run in opposite directions to their Flash counterparts _y and _rotation. Consequently, they must be inverted.

Converting from Flash to Cartesian Coordinates

Converting coordinates in the reverse direction, from Flash to Cartesian, is easy: the equations are simply reversed. Once again, x is assigned straight across, while y and rotation are multiplied by -1. This is how the procedure looks in ActionScript:

```
// convert Flash coordinates to Cartesian coordinates
// x, y, and rotation are existing number variables
// and mc is a movie clip instance
x = mc._x;
y = -mc._y;
rotation = -mc._rotation;
```

Polar Coordinates

In a two-dimensional world, there are two fundamentally different approaches to defining a location. The first, as we've seen, is to give the horizontal and vertical distances *x* and *y* from the origin (0, 0). For instance, we would say, "Point A is at (3, 4)," which means "Point A is three units to the right and four units up from (0, 0)." Thus, Cartesian coordinates are defined on a rectangular grid.

However, the same point in space can be defined in a different way, using a circular grid (see Figure 3-10). The circles on the grid correspond to the distance from the origin. Under this system, Point A in our example could be described as "5 units away from the origin, at 53.13 degrees."

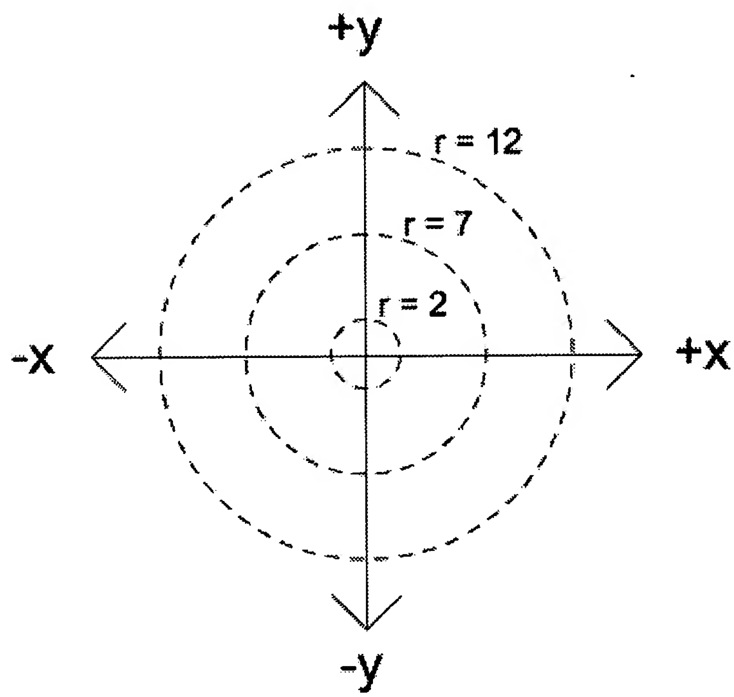


Figure 10: A circular grid of distances from the origin

When a distance and angle are used together to define a location, they are called *polar coordinates*. The notation for polar coordinates is (r, θ). The r coordinate is the point's distance or "radius" from the origin. The θ coordinate specifies the direction of the point from the origin, "as the crow flies."

Though less familiar than the Cartesian system, polar coordinates are extremely useful. For certain situations, *x* and *y* coordinates do not work as well as distance and angle. An airplane pilot, for example, needs to know the heading and distance of his destination. It's true that his start and end positions are defined on a semi-rectangular grid, in terms of longitude and latitude, which are similar to *x* and *y* values. Nevertheless, longitude and latitude do not *automatically* tell the pilot which direction to fly the plane, or how far away the destination is. Longitude and latitude must be converted into angle and distance; someone has to convert between coordinate systems.

Note: The pilot's situation is actually more mathematically complex than my simplified example. The surface of a globe such as the earth is measured with *spherical coordinates*. In this system, a point in 3D space is defined by (r, θ, f)—*rho*, *theta*, *phi*. In a standard mathematical model, r would be the radius of the earth, θ the angle of longitude, and f would be the angle from the North Pole. Another three-dimensional system is *cylindrical coordinates*, which define a location using (r, θ, z). I have used both of these systems in two of my experimental Flash pieces—cylindrical coordinates in "Photon Ring," and spherical coordinates in "Sphere Burst" (displayed at www.robertpenner.com).

Converting from Cartesian to Polar Coordinates

How does one take a point in Cartesian coordinates—an (*x*, *y*) pair—and convert it to the equivalent polar coordinates (*r*, *θ*)?

(4)Finding r and q Manually

You can do this conversion manually by plotting the point, for example, (3, 4), on graph paper. Then you can draw a line *r* from the origin to the point (3, 4). (See Figure 3-11.) You can now find the value of *r* by measuring its length with a ruler. You would find the distance to (3, 4) is about 5 units. To find *q*, you can use a protractor to measure the angle between the line to (3, 4) and the x-axis. You would find the angle to be about 53 degrees.

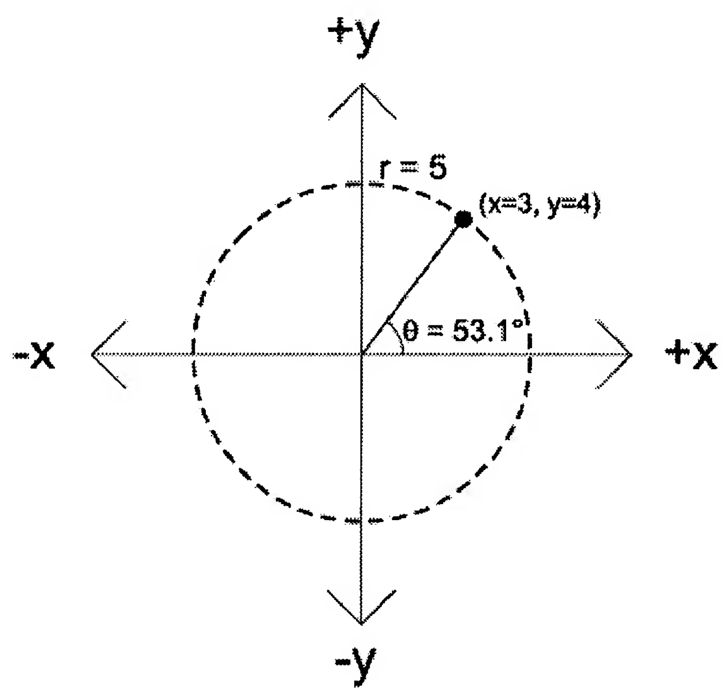


Figure 11: Polar coordinates *r* and *θ* for the point (3, 4)

This manual approach helps form an intuitive understanding of polar coordinates. However, it doesn't help us very much in a programming setting. We want to be able to convert between these coordinate systems dynamically, finding precise values with ActionScript.

(4)Finding r Mathematically

Thankfully, mathematicians figured out a way to find *r* long ago. The line *r* is the hypotenuse of a right triangle with sides *x*, *y*, and *r*. Thus, we can calculate *r* using the Pythagorean Theorem:

r = sqrt(x^2 + y^2)

If we pick the point (3, 4), this is how we use the equation to find the point's *r* value:

r = sqrt(3^2 + 4^2)
r = sqrt(9 + 16)
r = sqrt(25)

$r = 5$

The value of *r* shows that the point (3, 4) is 5 units from the origin.

(4)Finding q Mathematically

If we have a point (*x*, *y*), we can find the point's angle *q* by dividing *y* by *x* and taking the inverse tangent:

$q = tan^{-1}(y / x)$

We can take our familiar point (3, 4), and plug the *x* and *y* values into our equation to find *q*:

$q = tan^{-1}(4 / 3)$
$q = 53.1301...$

With the core mathematical equations for *r* and *q* under our belts, we are now just a step away from performing these calculations in Flash.

(4)Finding r and q with ActionScript

In ActionScript, we can use the following syntax to perform the conversion from the Cartesian coordinates (*x*, *y*) to the polar coordinates (*r*, *q*):

```
// convert Cartesian coordinates to polar coordinates

radius = Math.sqrt (x*x + y*y);
theta = Math.atan2 (y, x);
```

Note that *q* will be calculated in radians in this code. If degrees are preferred, we can use my custom function *Math.atan2D()*, discussed earlier in this chapter:

```
theta = Math.atan2D (y, x); // theta in degrees
```

Converting Cartesian to polar coordinates is a task we will need to perform repeatedly. It would be smart to encapsulate this process in a function. The most important features of a function are its input and output values. In this case, our input values are (*x*, *y*) and our output values are (*r*, *q*). The tricky part here is that a function can only return one value. How can we use one function to give us two values, *r* and *q*?

(4)Using a Point Object

The solution is to use a *point object* to store the coordinates for one point. So far, we have stored *x* and *y* in separate variables. What we can do instead is create a generic ActionScript object and give it *x* and *y* properties. Here is the basic procedure:

```
var p = new Object(); // create an object to represent a point
p.x = 3;  // assign a property for x
p.y = 4;  // assign a property for y
```

We can also use a shorthand notation to create the point object:

```
var p = { x:3, y:4 };
```

Note: The curly brace {} syntax is called the *anonymous object constructor.*

We can access the *x* and *y* values of the point object like this:

```
trace (p.x); // output: 3
trace (p.y); // output: 4
```

This point object is a convenient little package. It binds *x* and *y* together, and can be easily passed to and from functions. A point is really a single entity, so it makes sense to create a single object to represent it. Manipulations of points in code can be cleaner and easier with this approach. Chapters 4 and 5 will extend this concept. There, we will create custom *vector* objects which represent points in a similar way, but with additional functionality.

(4)A cartesianToPolar() Function

We have covered the basic code that will convert Cartesian coordinates to polar coordinates. We have also prepared a special point object. Now we're ready to create a custom function that encapsulates the conversion process. Here's how the function looks:

```
Math.cartesianToPolar = function (p) {
   var radius = Math.sqrt (p.x*p.x + p.y*p.y);
   var theta = Math.atan2D (p.y, p.x);
   return {r:radius, t:theta};
};
```

The *Math.cartesianToPolar()* function accepts one parameter: a point object p, which contains *x* and *y* properties. Using p.x. and p.y., the point's radius and theta are calculated. These values are assigned to the temporary variables radius and theta, respectively. In the last line of the function, a new object is created with two properties. The object's r property is assigned the value of radius, and the t property is assigned the value of theta. The function concludes by returning this new object to the outside world. Thus, when you call *Math.cartesianToPolar()*, you need to assign the result to a variable.

Here is an example of how to create a point in Cartesian coordinates and convert it to polar coordinates:

```
// test Math.cartesianToPolar()
var cartesianPt = { x:3, y:4 };
var polarPt = Math.cartesianToPolar (cartesianPt);
trace (polarPt.r); // output: 5
trace (polarPt.t); // output: 53.1...
```

In the example, a point is created at the coordinates (3, 4). After converting the point to polar coordinates, we find that the point is 5 units from the origin, at approximately a 1-degree angle.

Notice that *Math.cartesianToPolar()* calculates the angle in degrees. I used my custom function *Math.atan2D()*, discussed earlier in this chapter, because I prefer to think in degrees. If you want to find theta in radians instead, you can easily modify *Math.cartesianToPolar()* by replacing Math.atan2D with Math.atan2.

Converting from Polar to Cartesian Coordinates

To convert from polar to Cartesian coordinates, we start with *r* and *q*, and convert them to *x* and *y*. The basic equations for this conversion as follows:

$x = r \cos q$
$y = r \sin q$

This process is sometimes called "splitting/resolving into *x* and *y* components," especially when done with vectors (discussed in Chapter 4).

In ActionScript, these equations translate to the following code:

```
// convert from Polar coordinates to Cartesian coordinates
// theta is in degrees
x = radius * Math.cosD (theta);
y = radius * Math.sinD (theta);
```

Here is the *Math.polarToCartesian()* function, which converts polar coordinates to Cartesian:

```
Math.polarToCartesian = function (p) {
   var x = p.r * Math.cosD (p.t);
   var y = p.r * Math.sinD (p.t);
   return { x:x, y:y };
};
```

Similar to *Math.cartesianToPolar()*, the *Math.polarToCartesian()* function takes a single parameter: a point object. In this case, the object contains *r* and *t* properties, representing the point's radius and theta. These values are fed into the trigonometric equations to calculate *x* and *y* values. The function concludes by wrapping *x* and *y* into a new object, and returning it. Once again, I use the custom functions *Math.cosD()* and *Math.sinD()* so that we can specify angles in degrees.

Here is an example of how to create a point in polar coordinates and convert it to Cartesian coordinates:

```
// test Math.polarToCartesian()
var p = { r:5, t:90 };
var c = Math.polarToCartesian (p);
trace (c.x); // output: 3.061515884555594e-16 (approx. 0)
trace (c.y); // output: 5
```

In this example, a point is created that is 5 units from the origin, at a 90-degree angle. After converting the point to Cartesian coordinates, we find that the point is at (*x*, *y*) = (0, 5)—five units straight up the y-axis, as we would expect.

Conclusion

This chapter has been an introduction to trigonometry and coordinate systems. The concepts discussed in these two areas are crucial to understanding how objects relate within space. Our exploration has been both conceptual and practical, placing mathematical definitions and ActionScript side by side. Already, we have amassed a library of encapsulated and optimized functions, along with code examples. As it stands, this code can be put to many different uses, especially in projects that involve ActionScripted animation. In the next chapter, we will build on this foundation and take these concepts to another level. We will explore vectors: their qualities, relationships, and practical benefits for animation, physics, and dynamic design.

Document

document 4 of 6 Order Document

INSPEC - 1969 to date (INZZ)

Accession number & update
1487616, C80010790; 800000.

Title
Digital image shape detection.

Author(s)
Hord-R-M.

Author affiliation
Inst for Advanced Computation, Alexandria, VA, USA.

Source
AFIPS Conference Proceedings, vol.48. 1979 National Computer Conference, New York, NY, USA, 4-7 June 1979, p.243-54.
Sponsors: AFIPS.
Published: AFIPS, Montvale, NJ, USA, 1979, xi + 1095 pp.

Publication year
1979.

Language
EN.

Publication type
CPP Conference Paper.

Treatment codes
T Theoretical or Mathematical.

Abstract
Shape detection has been a topic of interest to the digital image pattern recognition community for many years, and it remains a prominent topic of discussion. The Power Spectrum of an image is well known to be independent of translation. The Mellin Transform has been shown to be scale-independent. The **Polar-Cartesian** (POL-CAR) Transform described **converts** rotation into translation. The Power Spectrum of the POLCAR Transform is then independent of rotation. Hence, a combination of these performed successively will allow shape to be matched to shape, independent of translation, rotation and scale. (10 refs).

Descriptors
pattern-recognition.

Keywords
shape detection; POLCAR Transform; translation; rotation; scale; digital image.

Classification codes
B6140C (Optical information processing).
C1250 (Pattern recognition).

save locally as: PDF document include search strategy

previous document next document order

Document

Select the documents you wish to save or order by clicking the box next to the document, or click the link above the document to order directly.

document 3 of 6 Order Document

INSPEC - 1969 to date (INZZ)

Accession number & update
3455901, B89061576; 890000.

Title
Soft X-ray lenses with 400AA outer zone width for nanostructure imaging.

Author(s)
Vladimirsky-Y; Kern-D-P; Meyer-Ilse-W; Greinke-B; Guttmann-P; Rishton-S-A; Attwood-D.

Author affiliation
Center for X-ray Opt, California Univ, Berkeley, CA, USA.

Source
14th International Conference on Microlithography/Microcircuit Engineering 88, Vienna, Austria, 20-22 Sept. 1988.
In: Microelectronic-Engineering (Netherlands), vol.9, no.1-4, p.87-8, May 1989.

CODEN
MIENEF.

ISSN
ISSN: 0167-9317, CCCC: 0167-9317/89/ ($3.50).

Publication year
1989.

Language
EN.

Publication type
CPP Conference Paper, J Journal Paper.

Treatment codes
A Application; P Practical.

Abstract
Fresnel zone plate lenses designed for imaging and microscopy at soft X-ray wavelengths have been fabricated with outer zone widths of 400AA. The zone plates were fabricated by electron beam lithography using a high resolution vector scan system designed specifically for nanometer pattern writing and equipped with high speed **polar** to **cartesian converter** for circular pattern generation. Gold electroplating was used to form the absorber pattern. Special measures taken to assure pattern fidelity are described. (0 refs).

Descriptors
integrated-circuit-technology; VLSI; X-ray-lithography.

Keywords
outer zone width; nanostructure imaging; Fresnel zone plate lenses; soft X ray wavelengths; electron beam lithography; vector scan system; pattern writing; circular pattern generation; absorber pattern;

pattern fidelity; 400 AA.

Classification codes

B2550G (Lithography).

B2570 (Semiconductor integrated circuits).

Numerical indexing

size: 4.0E-08 m.

save locally as: PDF document include search strategy

previous document next document order

Document

Select the documents you wish to save or order by clicking the box next to the document, or click the link above the document to order directly.

save locally as: PDF document include search strategy

previous document next document order

Fulltext-Link: IEEE

document 2 of 6 Order Document

INSPEC - 1969 to date (INZZ)

Accession number & update
4088255, B9203-1265F-069; 920212.

Title
A 540-MHz 10-b **polar-to-Cartesian converter.**

Author(s)
Gielis-G-C; van-de-Plassche-R; van-Valburg-J.

Author affiliation
Philips Res Lab, Eindhoven, Netherlands.

Source
IEEE-Journal-of-Solid-State-Circuits (USA), vol.26, no.11, p.1645-50, Nov. 1991.

CODEN
IJSCBC.

ISSN
ISSN: 0018-9200, CCCC: 0018-9200/91/1100-1645 ($01.00).

Publication year
1991.

Language
EN.

Publication type
J Journal Paper.

Treatment codes
X Experimental.

Abstract
A 10-b **polar-to-Cartesian converter** for generating digital sine and cosine waveforms simultaneously with a maximum sample rate of 540 MHz is presented. The **converter** is derived from a coordinate rotation digital computer (CORDIC) processor. Implementation details and the chip layout are given. The **converter** is implemented in a 1- mu m 13-GHz triple-level interconnect bipolar process, requiring 1000 mW from a single 5-V supply. The die size is 25 mm/sup 2/. (2 refs).

Descriptors
bipolar-integrated-circuits; convertors; data-conversion; microprocessor-chips; VLSI.

Keywords
CORDIC processor; **polar** to **Cartesian converter;** sample rate; coordinate rotation digital computer; chip layout; triple level interconnect bipolar process; single 5 V supply; die size; 10 bit; 1 micron; 540

MHz; 13 GHz; 5 V; 1 W; 6.4 mm.

Classification codes

B1265F (Microprocessors and microcomputers).
B2570B (Bipolar integrated circuits).
B1265Z (Other digital circuits).

Numerical indexing

frequency: 5.4E+08 Hz, 1.3E+10 Hz;
power: 1.0E+00 W;
size: 1.0E-06 m, 6.4E-03 m;
voltage: 5.0E+00 V;
word length: 1.0E+01 bit.